Universitat Politècnica de Catalunya

Trabajo de fin de grado

Desarrollo de compañero para Roguelike mediante técnicas de aprendizaje automático

Enrique Alexandre González Sequeira

Director: *Javier Béjar Alonso*

Departamento: Ciencias de la computación

2018

Resumen

La inteligencia artificial ha conseguido dominar varios juegos con las técnicas del aprendizaje por refuerzo, pero estos juegos acostumbran a ser de un solo jugador con un solo agente. En este proyecto se coge el *roguelike*, *Pixel Dungeon*, y se modifica para convertirlo en cooperativo. A partir de este, se desarrollan agentes de aprendizaje por refuerzo para los dos jugadores en juego, intentando conseguir que cooperen para vencer, identificando las dificultades más importantes y sacando conclusiones sobre los enfoques más adecuados para resolver el problema.

Resum

L'intel·ligència artificial ha aconseguit dominar diversos jocs amb tècniques d'aprenentatge per reforç, però aquests jocs acostumen a ser per un sol jugador amb un sol agent. En aquest projecte s'agafa un *roguelike*, *Pixel Dungeon*, i es modifica per tal de convertir-lo en cooperatiu. Partint d'aquest joc, es desenvolupen agents de aprenentatge per reforç per els dos jugadors en el el joc, intentant aconseguir que cooperin per vèncer, identificant les dificultats més importants i traient conclusions sobre l'enfoc més adequat per resoldre el problema.

Abstract

Artificial intelligence has managed to conquer a variety of different games with reinforcement learning techniques, but this games usually are single player games with a single agent. On this project *roguelike* game *Pixel Dungeon* is chosen and modified it to become a cooperative game. From this point, reinforcement learning agents will be developed for the two players on the game, making them cooperate to master the game and draw some conclusions on the best approaches to solve the problem.

Agradecimientos

Me gustaría dar las gracias a:

Mi tutor de este trabajo y profesor de aprendizaje automático, Javier Béjar, por guiarme y resolverme las dudas que se me planteaban a lo largo del proyecto.

A mi madre y a mi hermana por su incansable apoyo durante estos meses durante los buenos y los malos momentos.

A mi padre por darle optimismo a todas las cosas y por sus grandes consejos a lo largo de la carrera.

A Laura por apoyarme cada uno de los días y por darme la fuerza para trabajar cada día.

A Marc por todos estos años descubriendo la inteligencia artificial y peleando por nuestros objetivos.

Gracias.

Índice general

Capítulo 1

Contexto y alcance del proyecto

1.1 Contexto y formulación del problema

La inteligencia artificial es un campo que ha tenido un gran crecimiento en los últimos años hasta el punto de convertirse en una de las ramas principales de la computación. Actualmente se han conseguido logros inimaginables en el pasado y que la sociedad aprovecha de manera significativa hoy en día.

A su vez la inteligencia artificial siempre ha estado muy ligada a los juegos y especialmente a los videojuegos. Debido a que estos son un dominio ideal para hacer pruebas en IA por la facilidad que ofrecen en la interacción persona-máquina. En un videojuego el jugador tiene varios comandos para interactuar con el entorno y dependiendo del videojuego se tienen más o menos posibles interacciones y de distintas complejidades. Por eso mismo en un videojuego las opciones son limitadas y son un campo seguro para la prueba de inteligencias artificiales.

Además, hoy en día la industria de los videojuegos ha crecido de manera notoria y mientras antes se creaban unos pocos juegos al año hoy en día se crean miles de videojuegos al año de distintos géneros con complejidades y retos distintos para las inteligencias artificiales. Hoy en día los videojuegos son accesibles para todo el mundo en múltiples plataformas lo que ha facilitado aun más el uso de los mismos para inteligencias artificiales.

Y con la creación de juegos más complejos y gigantes las grandes compañías recogen una gran cantidad de datos sobre su juego mediante técnicas de *big data* o *data mining research* que pueden ser usados por las inteligencias artificiales que se basan en *machine learning*. Por ejemplo, para *Call of Duty Black Ops 4* (Treyarch, 2018) se han usado datos recolectados de todos sus anteriores juegos durante sus tres años de desarrollo para mejorar la experiencia y crear una nueva inteligencia artificial (IA) usando *machine learning* para su complejo modo de zombis.

Hoy en día los juegos más jugados del mundo son juegos en línea que permiten enfrentar a distintos jugadores y que compitan entre ellos. La competición es lo que los hace tan populares y en este aspecto la inteligencia artificial también se puede favorecer, puesto que gracias a los grandes avances en este campo se han conseguido crear inteligencias artificiales capaces de vencer de manera contundente a un ser humano experto en diversos juegos, ya sean de mesa o de videojuegos.

Por ejemplo, en 1996 **Deep Blue** ganó al Gran Maestro *Gary Kasparov* en ajedrez. En 2016 *Google Deepmind* puso a su inteligencia artificial **AlphaGo**[**?**] a jugar contra un famoso jugador profesional de Go coreano *Lee Sedol* con un premio de 1 millón de dolares para el ganador, *AlphaGo* se hizo con la victoria por 4-1. Por último, en 2017 *OpenAI* creó una inteligencia artificial capaz de ganar al jugador profesional de Dota2 Danylo *"Dendi"* uno contra uno. Y en 2018 *OpenAI* ha mejorado la IA haciéndola capaz de trabajar en equipo y ha conseguido vencer a un equipo de jugadores amateur sin muchas dificultades, el objetivo para finales de este año es ser capaz de vencer a un equipo profesional de Dota2.

En este proyecto se aplicarán técnicas de aprendizaje por refuerzo para crear una IA, pero en vez de controlar al personaje principal del videojuego lo que hará es controlar a un compañero que debera ayudar al jugador aprendiendo del mismo. Para este tipo de proyecto el director y yo acordamos que el género de videojuego idóneo para este proyecto es el género *Roguelike* por los motivos que se explican a continuación.

1.1.1 Roguelike

Roguelike o **exploración de mazmorras** es un género de videojuegos orientado a que el jugador se adentre en una mazmorra en la que se suelen crear los niveles de la misma de manera procedural. Suelen ser juegos de un jugador con un sistema por turnos o microturnos: hasta que el jugador no realiza una acción, el entorno no reacciona.

El objetivo de estos juegos es muy simple: llegar al último nivel de la mazmorra, derrotar al jefe final y regresar. Una de las características fundamentales en estos juegos es que la muerte es permanente, una vez el jugador muere se termina el juego y se tiene que empezar desde el principio. Este es uno de los factores por los que la dificultad de este tipo de juegos es muy elevada.

El juego recompensa al jugador al recoger objetos que lo hacen más poderoso pero, estos objetos suelen estar rodeados de trampas mortales. De esta manera los enemigos a los que se va enfrentando el jugador son cada vez más poderosos, igual que lo es el jugador. Ya que suelen tener un gran nivel de dificultad, el hecho de introducir un compañero que te ayude se convierte en un reto muy interesante. Puesto que si la IA actua mal podría acabar perjudicando al jugador en vez de ayudarlo. A cambio, esta ayudará al jugador a recoger objetos y volverse más

poderoso al igual que ayudarle a atacar a enemigos. Como el juego es por turnos se trata de un entorno ideal para nuestra IA.

De entre todos los juegos de este género que existen se quería un juego *open source* que se ejecutase en pc y que fuera visualmente sencillo de entender. Existen una decena de juegos open source que cumplen los requisitos anteriores y se ajustan a los objetivos de este trabajo, pero al final el juego elegido ha sido *Pixel Dungeon* más concretamente un mod del mismo *Shattered Pixel Dungeon*.

1.1.2 Pixel Dungeon

Pixel Dungeon es un juego *open source* del género roguelike tradicional con gráficos *pixel art* y una interfaz simple con versión para Android, iOs, Steam , y una versión de escritorio que es la que se usará en el proyecto.

Pixel Dungeon ha sido completamente desarrollado por *Watabou* y inspirado por *Brogue* un popular juego del género.

El juego consta de 4 héroes y consiste en explorar las profundidades de *Pixel Dungeon*, recolectar objetos para volverte más poderoso y combatir enemigos para encontrar el *Amuleto de Yendor*. El juego tiene 26 niveles de profundidad.

1.1.3 Shattered Pixel Dungeon

Shattered Pixel Dungeon es un mod de *Pixel Dungeon* que empezó como un proyecto para mejorar el juego original y ha evolucionado a un juego completo con características únicas. Este mod incluye más objetos, más enemigos y más artefactos pero la razón por la que se ha escogido este juego en vez del original es porque el repositorio público de este juego daba más facilidades a ser modificado que el juego original. Igual que el juego original, el juego está hecho en Java y se usará la versión de escritorio con gdx.

1.1.4 Aprendizaje por refuerzo

Es un área del aprendizaje automático inspirada en como aprendemos los seres vivos, a traves de recompensas positivas o negativas dependiendo de las acciones que se realicen. Un agente se encuentra en un **instante** t, en un **estado** s y realiza una **acción** a de entre todas las acciones posibles en el estado actual. El entorno reacciona a esta acción y **otorga** una **recompensa** r. Tras hacer este proceso múltiples veces, el agente aprende a seleccionar las acciones que maximizan su suma de recompensas a largo plazo.

El objetivo del agente es encontrar una **política** π ideal que seleccione las acciones óptimas para cada estado que maximizaran la recompensa obtenida a largo plazo. Para ello, se necesita una **función de valor** v que estime la recompensa esperada a largo plazo teniendo en cuenta un **factor de descuento** γ(valor entre 0 y 1) que sirve para dar más o menos importancia a las acciones lejanas en el tiempo, en el momento de crear la política π.

Esto se consigue modelando el entorno como un Problema de Decisión de Markov (MDP), de tal manera que para cada estado hay una transición hacia otro estado asociada a una acción posible desde ese estado con una probabilidad dada. Y cada transición otorga al agente una recompensa asociada a la misma.

1.2 Actores implicados

1.2.1 Desarrollador del proyecto

En este proyecto el único desarrollador es Enrique Alexandre González Sequeira, estudiante de ingeniería informática en la Facultad de Informática de Barcelona (UPC). Las funciones del desarrollador del proyecto son: investigar, desarrollar y documentar todo el proyecto incluido todo el software que sea requerido. Además, debe organizar todas las tareas a realizar en el proyecto. Este actor debe cumplir los plazos estipulados con el director además de estar comunicados en todo momento.

1.2.2 Director del proyecto

El director de este proyecto es Javier Béjar Alonso, profesor del departamento de Ciencias de la Computación en la Facultad de Informática de Barcelona (UPC) y miembro de `Grup d´Enginyeria del Coneixement i Aprenentatge Automàtic (KEMLG)`. Es el principal responsable guiando, dando consejo y en general ayudando al desarrollador. Su trabajo es ayudar al desarrolador con el enfoque y ejecución del proyecto contribuyendo a solventar los obstáculos que se puedan aparecer en el camino.

1.2.3 Beneficiarios

Los beneficiarios directos de este proyecto son las empresas de desarrollo de videojuegos. Estas podrán aprovechar las técnicas descritas en este proyecto para usarlas para sus propios videojuegos e incluso motivarlas a incluir compañeros en un videojuego que aprenda con las acciones del jugador. Otros beneficiarios indirecto son los propios jugadores que mejorarán su experiencia de juego y disfrutarán

de un compañero que actuará según sus propias acciones. De esta manera, podrán ver reflejada su experiencia de juego en el compañero.

1.3 Estado del arte

Como se comentaba anteriormente, la inteligencia artificial ha conseguido grandes logros a lo largo de su historia usando como entorno los videojuegos. Por ejemplo, Deepmind [2][3], empresa adquirida por *Google* en 2014 y pionera y experta en aprendizaje por refuerzo consiguió en 2015 crear una IA con un nivel superior a un jugador humano en varios juegos Atari 2600 [9][10]. Para conseguir tal azaña se usaron algoritmos de `Deep Reinforcement Learning` como Deep Q-Network (DQN) algoritmos que se usarán como base de este proyecto. Para aprender a jugar a este tipo de juegos la única información que se usaba eran los píxeles de la pantalla como input.

Uno de los últimos logros en inteligencia artificial aplicado a videojuegos es el mencinado anteriormente OpenAI de Elon Musk, capaz de vencer a un jugador profesional de Dota2 en un uno contra uno en el `The International 2017` [6]. Y recientemente en el mismo evento de 2018 se logró vencer a un equipo de profesionales y exprofesionales del juego [7][8]. Aunque esto pueda parecer el futuro, y que si una IA es capaz de jugar a un juego tan complejo como Dota2 de manera tan parecida a un jugador profesional deberían ser capaces de resolver problemas muy complejos, la verdad es que tiene muchas limitaciones. Como explican en OpenAI [11], para llegar a un nivel de juego tan alto se han usado 100.000 procesadores para entrenar a los bots, una potencia de computo millonaria y solo al alcance de las grandes compañías del sector. Además, para simplificar un poco la gran dificultad del juego y hacer más "justo" el enfrentamiento se limitaron algunas características del juego y se usaron los mismos "héroes" en ambos bandos.

Todo esto nos lleva a creer que las técnicas actuales de Deep Reinforcement Learning requieren una cantidad de tiempo y hardware abismales para llegar a solventar problemas de una complejidad similar a jugar a Dota2 a nivel profesional, por lo tanto problemas más difíciles serían muy costosos de abordar. Aún deben avanzar las técnicas de *Deep Reinforcement Learning* para conseguir algoritmos más eficientes.

Como comenta David Silver(Deepmind) en su curso de Reinforcement Learning [12] se ha conseguido utilizar Reinforcement Learning para que un helicoptero en miniatura aprenda a volar de forma autónoma en condiciones ideales. El siguiente paso en este campo es conseguirlo con uno de tamaño real y ser capaz de que cualquier vehículo sea capaz de controlarse a si mismo. Esto es un objetivo muy ambicioso y aún queda mucho para que se haga real, pero hay pequeños experimentos como el antes mencionado, que nos acercan más al límite del *Reinforcement Learning*.

Aunque el rendimiento que ha conseguido Deepmind en los juegos de Atari 2600 mediante *Deep Reinforcement Learning* es impresionante, no dejan de ser juegos muy antiguos con posibilidades muy limitadas debido a su fecha de publicación, además, son juegos basados en el comportamiento del jugador principal. En este proyecto se quiere intentar dar un paso más y estudiar como se podrían utilizar los conocimientos y técnicas que nos han dejado *Deepmind* entre otros para crear una inteligencia artificial para un compañero del jugador protagonista en un videojuego.

Por todo lo explicado anteriormente se cree conveniente usar como base los conocimientos dados por *Deepmind* o *OpenAI* pero, dado que en este proyecto no se dispone de los recursos antes mencionados se tendrá que dar un pequeño cambio en el enfoque. En vez de usar como input todos los píxeles de la pantalla se intentará utilizar solo una parte relevante de la misma y de esta manera evitar información redundante. Al ser un juego *Roguelike* por turnos, este tiene una parte de la interfaz donde se escriben los eventos relevantes que acontecen en el juego. Esta interfaz es una pequeña parte de todos los píxeles de la pantalla, pero de donde más información se puede extraer.

1.4 Alcance del proyecto

Los requisitos para el desarrollo de este proyecto son un juego open source de estilo *roguelike* (Pixel Dungeon) para poder modificarlo y añadirle un compañero. Además, se usará *Tensoforce* una librería de Python dedicada a aprendizaje por refuerzo por lo que se necesita instalar y configurar.

Después de esto se necesita escoger un método para conseguir entrenar al compañero junto al jugador protagonista. Como se ha comentado en apartados anteriores la información con la que trabaja una IA son los píxeles de la pantalla que se recogerán como input a medida que se juega al juego.

El objetivo inicial sería que el compañero del jugador fuese capaz de actuar de forma útil sin llegar a estorbar al jugador, luego se intentará que su nivel de juego vaya creciendo y idealmente la meta sería que el compañero fuera una ayuda relevante para el jugador, cogiendo objetos y atacando en los momentos oportunos. El objetivo es que el jugador sienta que el nivel de dificultad del juego es menor si utiliza al compañero.

1.5 Posibles obstáculos y problemas

En este apartado analizaremos los posibles obstáculos que puedan surgir durante la realización del proyecto y posibles soluciones a los mismos.

1.5.1 Dificultad del problema y bugs

Aunque el juego escogido no es de una complejidad muy elevada, el reto de este proyecto no es para nada trivial. Requerirá de esfuerzo encontrar los algoritmos adecuados para lograr el objetivo. Debido a la complejidad del software es sencillo que aparezcan bugs cuando se modifique el código fuente. Para asegurarnos de que no existan bugs se realizaran diversos tests que verifiquen que el sistema funciona correctamente.

1.5.2 Calendario y planificación

Debido a que el tiempo para realizar un proyecto de esta magnitud es bastante escaso (4 meses), la planificación que se use es de vital importancia para cumplir los plazos impuestos por la universidad. Además, el tiempo requerido para entrenar un agente en deep reinforcement learning es elevado ya que, se debe entrenar varias veces para ajustar los parámetros.

1.5.3 Potencia de computo

Además de mucho tiempo, también se necesita mucha potencia de computación para lidiar con este problema. Se estudiarán dos posibles soluciones: utilizar una `NVIDIA GEFORCE GTX 970` o usar computación en la nube. Se estudiará cuál es la mejor opción.

1.6 Metodología

Debido a que el tiempo de desarrollo es corto y se quiere feedback continuo de como va el proyecto en todo momento, para el desarrollo de este proyecto se utilizarán técnicas de la metodología Agile. La metodologia Agile otorga flexibilidad, un rápido desarrollo y resultados inmediatos en menos tiempo que otras metodologías. Ya que este tipo de metodología esta pensada básicamente para trabajar en equipo con otros compañeros, no se seguirá la metodología al pie de la letra. Pese a ello, se usarán los conceptos que pueden servir en el desarrollo de un proyecto en solitario.

1.6.1 Ciclos de desarrollo cortos

Se usarán ciclos de desarrollo cortos con fechas límite cada semana para tener constancia del estado del proyecto en tiempo real.

1.7 Herramientas

1.7.1 Herramientas de seguimiento

- **Trello:** Trello es una herramienta web que te permite organizar de una forma muy clara y sencilla las tareas a realizar en un proyecto. Se pondrán tareas semanales que se deberan cumplir en el plazo acordado.
- **Github:** Para controlar la evolución del código se usará un reposito en Github.

1.8 Métodos de validación

Se harán comprobaciones de errores via *Github* y se harán reuniones con el director del proyecto para garantizar la validación de los objetivos.

Además, para comprobar si la inteligencia artificial se comporta de manera natural se comparará con la forma de jugar que tendría un humano.

1.9 Integración de los conocimientos

A continuación se explican algunos de los conocimientos adquiridos en algunas de las asignaturas de la carrera que se usan en el desarrollo de este trabajo final de carrera.

1.9.1 Inteligencia artificial (IA)

En esta asignatura es donde se me enseñaron las bases de que es una inteligencia artificial y como funcionan además de introducir algunos conceptos básicos que se usan en el agente.

1.9.2 Aprendizaje automático (APA)

En aprendizaje automático no se enseña nada de aprendizaje por refuerzo, pero se explican redes neuronals, que usan algunos de los métodos utilizados en este proyecto como por ejemplo DDQN.

1.9.3 Visión por computador(VC)

En esta asignatura se nos enseñan maneras de como tratar los píxeles de una imagen y se usan esos conocimientos para tratar los datos de los píxeles que lee el agente antes de aplicar las técnicas por refuerzo.

1.9.4 Videojuegos (VJ)

En esta asignatura se me enseñaron las bases de como programar un videojuego y la estructura que estos suelen tener, me ha servido para modificar el juego Pixel Dungeon para añadir al compañero. Gracias a los conocimientos adquiridos en esta asignatura fue mucho más fácil para mi el entender el código de un juego gigantesco y entender que función tiene cada clase para el funcionamiento del juego, además de entender cada capa de profundidad de su desarrollo.

Capítulo 2

Planificación del proyecto

2.1 Planificación y tiempos

La duración estimada de este proyecto es de 4 meses. Empezando a principios de septiembre de 2018 y terminando a finales de enero de 2019, antes de la exposición oral.

Hay que tener en cuenta que debido a la metodología que se usará (Agil) en el caso en que se retrase alguna tarea se cambiarán los plazos a medida que el proyecto evolucione.

2.2 Descripción de tareas

2.2.1 Realización de la fase inicial

Es la primera parte del proyecto a realizar y consta de las siguientes partes:

- Contextualización, formulación del problema, estado del arte y definición del alcance del proyecto
- Planificación temporal
- Gestión económica y sostenibilidad
- Módulo específico de especialidad: Computación

2.2.2 Aprender sobre aprendizaje por refuerzo

En el primer mes del proyecto se obtendrán más conocimientos sobre aprendizaje por refuerzo y estudiar algunas aplicaciones en los videojuegos. Para ello se analizarán diversos artículos sobre el tema, como por ejemplo el de A.I Wiki [1]. A continuación se profundizará el el estado del arte mediante el libro *Artificial Intelligence and Games* [4] ,recomendado por mi director en el que se trata de la estrecha relación entre la IA y los videojuegos. Además, se entra en detalle en algunos conceptos de aprendizaje por refuerzo.

Para aprender sobre los algoritmos que habrá que aplicar sobre aprendizaje por refuerzo, se utilizará los repositórios de Github [13] [14]. Un curso de Deepmind impartido por David Silver [12] y el libro *Reinforcement Learning: An Introduction* [5].Ambas fuentes de información recomendadas también por el director del proyecto.

2.2.3 Modificar Shattered Pixel Dungeon

Antes de empezar a implementar la IA se necesitará modificar el juego *Shattered Pixel Dungeon* para que cumpla los requisitos necesarios. El juego es *open source* y se puede conseguir en su pagina web. Al igual que la gran mayoría de *Roguelike*, el juego no consta de un compañero de base que ayude al personaje principal. Deberemos modificar el códifo fuente del juego y añadirlo. Para ello se deberá estudiar primero el funcionamiento del flujo del código e intentar reaprovechar todos los recursos que tiene el juego. Esta tarea se realizará en función de los requerimientos que la IA necesite del compañero. Además, es una tarea **clave** en el desarrollo del proyecto debido a que es la base de la IA que se quiere desarrollar.

2.2.4 Instalación y configuración del entorno

El primer paso en la implementación de la IA será la configuración del entorno en el que se trabajará. Para ello necesitaremos el ejecutable del juego ya modificado (.jar) y la librería de Python *Tensorforce*; una libreria de *TensorFlow* aplicada a *reinforcement learning*. Se instalará todo en Windows usando Pycharm, un IDE de Python. También se instalarán librerias que permitan hacer cálculos mediante la GPU para la parte del entreno de agentes.

Una vez instalado el entorno se necesitará un tiempo para familiarizarse con el mismo, haciendo diversas pruebas.

2.2.5 Buscar mejor técnica

Utilizando el entorno antes mencionado se realizarán las pruebas necesarias para determinar qué métodos son los más eficientes e interesantes para desarrollar los agentes más útiles del compañero.

2.2.6 Implementación, ejecución y testing de los agentes

Se implementarán y entrenarán a los agentes que se han decidido en la tarea anterior. Además se ejecutarán los resultados y se harán tests para estudiar qué parámetros se deben modificar.

2.2.7 Análisis y conclusiones

Se analizarán los resultados de los distintos agentes y su interacción con el personaje principal y se sacarán conclusiones sobre ellos, comparando el rendimiento de cada uno con el resto para comprobar cuáles responden mejor.

2.2.8 Redacción de la memoria anexo y documentación

Se anotará por escrito todo el proceso por el que ha pasado el proyecto, desde la implementación ,hasta el análisis y las conclusiones.

2.3 Duración aproximada de las tareas

Tarea	Duración estimada (h)
Realización de la fase inicial	90
Aprender sobre aprendizaje por refuerzo	100
Modificar Shattered Pixel Dungeon	50
Instalación y configuración del entorno	10
Buscar mejor técnica	25
Implementación, ejecución y testing de los agentes	170
Análisis y conclusiones	25
Redacción de la memoria anexo y documentación	40
Total	510

Cuadro 2.1: Duración estimada de las tareas en horas

2.4 Valoración de alternativas y plan de acción

Durante el desarrollo del proyecto surgirán variaciones en el plan que nos haran modificarlo. Para eso trabajamos con la metodología *agil* que nos permitirá tener constante *feedback* del proyecto y adaptar los tiempos de cada tarea dependiendo de la evolución del mismo. Por ejemplo, si una tarea se realiza antes del tiempo estimado, se empezará la siguiente inmediatamente.

El trabajo de fin de grado consta de 18 ECTS, lo que equivale a una dedicación aproximada de 30 horas a la semana, teniendo en cuenta que se dispone de unas 18 semanas para la realizaciónd de este trabajo. Hay un total de 540 horas para dedicarle al proyecto. Esto da unas 30 horas de margen en caso de imprevistos. De esta manera podemos garantizar que tenemos una cantidad de margen suficiente para poder acabar el proyecto a tiempo.

2.5 Posibles retrasos

Las principales posibles razones de retraso en el proyecto son el entrenamiento de los agentes debido a la potencia de computo necesaria y la modificación del juego. Por ello son las tareas que más tiempo requieren y se actualizará la planificación en función de como evolucione el proyecto.

En el caso en que la potenciá de cómputo sea un problema grave se cambiarán los recursos a computación en la nube.

2.6 Recursos

El principal recurso utilizado son los recursos humanos, la persona que realiza este proyecto tendrá que hacer todas las tareas antes explicadas en el tiempo previsto. Además se usarán recursos de hardware y software. Más adelante en este documento se explicarán detalladamente los costes exactos de todos los recursos utilizados.

2.6.1 Hardware

Las herramientas de *hardware* utilizadas serán: Un PC con las siguientes características:

- Intel I7 4790k

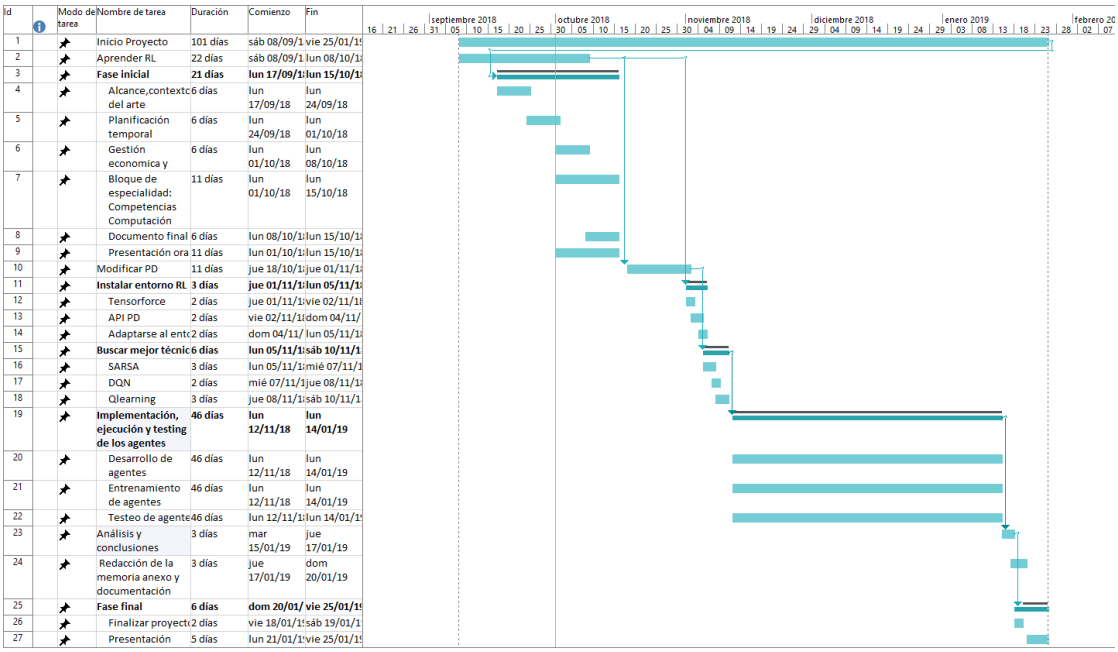
- Nvidia Geforce 970 GTX
- 16 GB ram

2.6.2 Software

Las herramientas *software* utilizadas serán:

- SerpentAI
- Python3
- Shattered Pixel Dungeon
- Latex
- Microsoft Project
- Trello

2.7 Diagrama de Gantt



Id	Modo de tarea	Nombre de tarea	Duración	Comienzo	Fin
1		Inicio Proyecto	101 días	sáb 08/09/1	vie 25/01/19
2		Aprender RL	22 días	sáb 08/09/1	lun 08/10/18
3		**Fase inicial**	**21 días**	**lun 17/09/18**	**lun 15/10/18**
4		Alcance,contexto del arte	6 días	lun 17/09/18	lun 24/09/18
5		Planificación temporal	6 días	lun 24/09/18	lun 01/10/18
6		Gestión economica y	6 días	lun 01/10/18	lun 08/10/18
7		Bloque de especialidad: Competencias Computación	11 días	lun 01/10/18	lun 15/10/18
8		Documento final	6 días	lun 08/10/18	lun 15/10/18
9		Presentación ora	11 días	lun 01/10/18	lun 15/10/18
10		Modificar PD	11 días	jue 18/10/18	jue 01/11/18
11		**Instalar entorno RL**	**3 días**	**jue 01/11/18**	**lun 05/11/18**
12		Tensorforce	2 días	jue 01/11/18	vie 02/11/18
13		API PD	2 días	vie 02/11/18	dom 04/11/
14		Adaptarse al entc	2 días	dom 04/11/	lun 05/11/18
15		**Buscar mejor técnic**	**6 días**	**lun 05/11/18**	**sáb 10/11/18**
16		SARSA	3 días	lun 05/11/18	mié 07/11/18
17		DQN	2 días	mié 07/11/18	jue 08/11/18
18		Qlearning	3 días	jue 08/11/18	sáb 10/11/18
19		**Implementación, ejecución y testing de los agentes**	**46 días**	**lun 12/11/18**	**lun 14/01/19**
20		Desarrollo de agentes	46 días	lun 12/11/18	lun 14/01/19
21		Entrenamiento de agentes	46 días	lun 12/11/18	lun 14/01/19
22		Testeo de agente	46 días	lun 12/11/18	lun 14/01/19
23		Análisis y conclusiones	3 días	mar 15/01/19	jue 17/01/19
24		Redacción de la memoria anexo y documentación	3 días	jue 17/01/19	dom 20/01/19
25		**Fase final**	**6 días**	**dom 20/01/**	**vie 25/01/19**
26		Finalizar proyect	2 días	vie 18/01/19	sáb 19/01/19
27		Presentación	5 días	lun 21/01/19	vie 25/01/19

Figura 2.1: Diagrama de Gantt del proyecto

2.8 Seguimiento de la planificación: Hito intermedio

Por lo general, aunque han habido bastantes imprevistos que han retrasado la realización de este proyecto, se ha seguido la planificación con rigor y actualmente sigue siendo viable terminar el proyecto segun la misma. En el punto actual del proyecto se han realizado las siguientes tareas en el tiempo que determinaba la planificación.

2.8.1 Tareas Realizadas

Fase inicial

Se realizó la fase inicial que corresponde a GEP a tiempo, se hicieron todas las entregas y presentaciones correspondientes.

Modificar Pixel Dungeon

Se modificó el juego Pixel Dungeon tal y como marcaba la planificación en el tiempo estipulado de 11 días. Esta fue una de las tareas más costosas en horas de programación debido a que se debía de entender un codigo Java muy extenso. Se añadió el compañero al juego capaz de moverse ortogonalmente y interactuar con la celda en la que se encuentre en un instante de tiempo. Para atacar a un enemigo el compañero se tiene que pegar a un enemigo y moverse en su dirección. Se explicará con más detalle en el documento final.

Instalar entorno RL, técnicas y implementación

Se han instalado las librerías a utilizar (SerpentAI y Tensorforce) y se han probado los métodos DDQN,DQN y Sarsa. Actualmente se está en la tarea "Implementación, ejecución y testing de los agentes" y en la realización de estas tareas es donde ha habido más desviaciones en el tiempo y costes por los distintos imprevistos.

2.8.2 Imprevistos y desviaciones de los costes

Cuando la tarea " Implementación, ejecución y testing de los agentes" fue empezada empezarón una cadena de acontecimientos que han hecho que el desarrollo de esta tarea se retrasase. Al empezar a implementar y entrenar los agentes me di cuenta que el espacio en disco y memoria ram no eran suficientes debido a múltiples errores por memoria al entrenar. Tras esto decidí adquirir un ssd de más capacidad y 8gb más de ram, esto supone un incremento en los costes y en el tiempo debido a que mientras no llegaba el hardware no podía entrenar. Al llegar se instaló el ssd sin problemas pero al instalar la memoria ram el PC tuvo serios problemas y dejó de funcionar. Esto produjo que perdiera la disponibilidad del PC durante una semana que estaría en reparación. En total perdí una semana de trabajo entrenando y 250 euros. Un tiempo preciado pero que por suerte, gracias a como está montada la planificación de tiempo y costes, se ha podido seguir con el proyecto adecuadamente aunque apretando un poco más en las semanas posteriores. Debido a que la tarea en la que me encontraba cuando surgieron los imprevistos era de larga duración (1 mes y medio) no se ha notado tanto en el seguimiento de la planificación.

2.8.3 Estado actual del proyecto y previsión

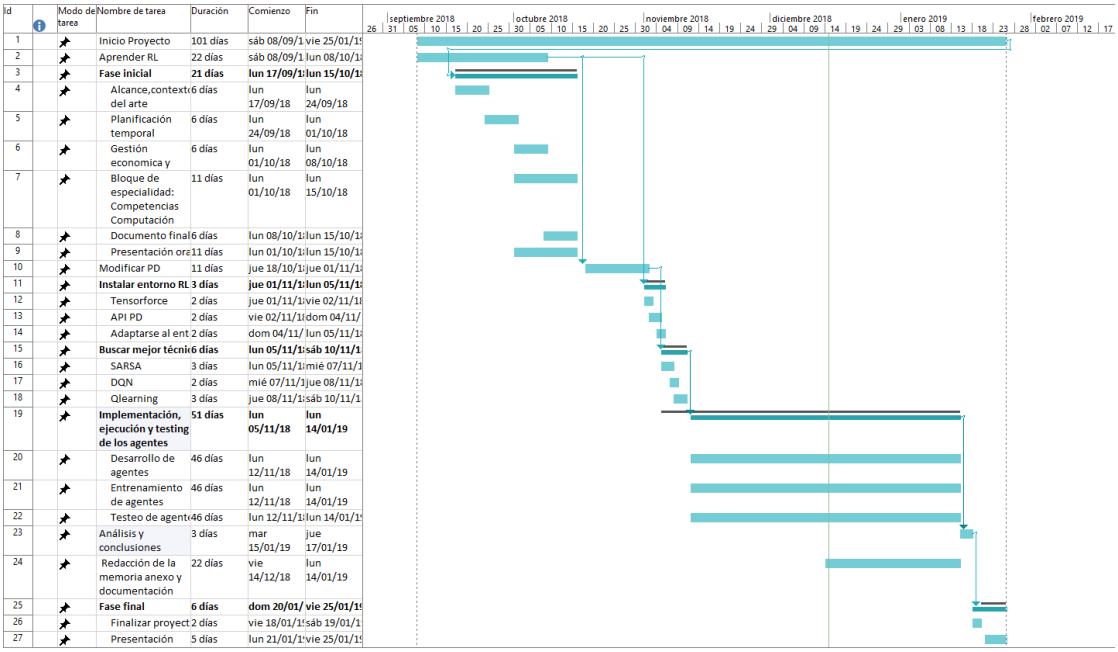
Como he mencionado anteriormente el proyecto se encuentra en la tarea de implementación y entrenamiento de los agentes, actualmente se tienen implementados y probados *DDQN* y *Sarsa* a falta de pequeñas modificaciones que se han comentado

con el tutor. Se quieren implementar tres métodos más *DQN*, *Q-learning* y *PPO*. *DDQN* es una mejora de *DQN* por lo que ya esta implementado y *Q-learning* tiene una fuerte similitud con *Sarsa*, por lo que el único método que queda por implementar desde cero es *PPO*. En las dos semanas aproximadamente se acabará de implementar y probar todos los métodos y se compararán los resultados mientras se redacta el documento final.

2.8.4 Cambios en la metodología

Se ha seguido trabajando con la metodología descrita en el documento de la fase inicial, no se ha modificado debido a que se trabajaba bien con dicha metodología y se han cumplido todos los plazos asignados en la planificación sin problemas.

2.9 Diagrama de Gantt final



Id	Modo de tarea	Nombre de tarea	Duración	Comienzo	Fin
1		Inicio Proyecto	101 días	sáb 08/09/1	vie 25/01/19
2		Aprender RL	22 días	sáb 08/09/1	lun 08/10/18
3		**Fase inicial**	**21 días**	**lun 17/09/1**	**lun 15/10/18**
4		Alcance,context del arte	6 días	lun 17/09/18	lun 24/09/18
5		Planificación temporal	6 días	lun 24/09/18	lun 01/10/18
6		Gestión economica y	6 días	lun 01/10/18	lun 08/10/18
7		Bloque de especialidad: Competencias Computación	11 días	lun 01/10/18	lun 15/10/18
8		Documento final	6 días	lun 08/10/1	lun 15/10/18
9		Presentación ora	11 días	lun 01/10/1	lun 15/10/18
10		Modificar PD	11 días	jue 18/10/1	jue 01/11/18
11		**Instalar entorno RL**	**3 días**	**jue 01/11/1**	**lun 05/11/18**
12		Tensorforce	2 días	jue 01/11/1	vie 02/11/18
13		API PD	2 días	vie 02/11/1	dom 04/11/
14		Adaptarse al ent	2 días	dom 04/11/	lun 05/11/18
15		**Buscar mejor técni**	**6 días**	**lun 05/11/1**	**sáb 10/11/1**
16		SARSA	3 días	lun 05/11/1	mié 07/11/1
17		DQN	2 días	mié 07/11/1	jue 08/11/18
18		Qlearning	3 días	jue 08/11/1	sáb 10/11/1
19		**Implementación, ejecución y testing de los agentes**	**51 días**	**lun 05/11/18**	**lun 14/01/19**
20		Desarrollo de agentes	46 días	lun 12/11/18	lun 14/01/19
21		Entrenamiento de agentes	46 días	lun 12/11/18	lun 14/01/19
22		Testeo de agent	46 días	lun 12/11/1	lun 14/01/19
23		Análisis y conclusiones	3 días	mar 15/01/19	jue 17/01/19
24		Redacción de la memoria anexo y documentación	22 días	vie 14/12/18	lun 14/01/19
25		**Fase final**	**6 días**	**dom 20/01/**	**vie 25/01/19**
26		Finalizar proyect	2 días	vie 18/01/1	sáb 19/01/1
27		Presentación	5 días	lun 21/01/1	vie 25/01/19

Figura 2.2: Diagrama de Gantt definitivo del proyecto

Capítulo 3

Gestión económica y sostenibilidad

3.1 Encuesta de sostenibilidad

Tras haber realizado la encuesta , me he dado cuenta de que no se da la educación ni la importancia suficiente a la sostenibilidad en la etapa universitaria. En los cuatro años de carrera solo se ha tocado la sostenibilidad de manera real en una asignatura (AC).

En cuanto al impacto medioambiental de un proyecto, creo que es el campo de la sostenibilidad que menos se tiene en cuenta en la etapa universitaria, los proyectos informáticos no producen gases ni emisiones contaminantes pero indirectamente pueden tener un impacto crucial en el medioambiente. Por ejemplo, los PC que se desechan si no son bien reciclados pueden llevarse a lugares donde intentan quemar el oro de manera ilegal y generando una cantidad de emisiones muy perjudical para el medio ambiente. Además, en el mundo real no se suele dar la importancia que merece y se da prioridad a la maximización de beneficios antes que al impacto ambiental. Desconozco los métodos y herramientas que se utilizan para que un proyecto tenga en cuenta el medioambiente.

En cuanto a la gestión económica, accesibilidad y seguridad de un producto creo que en la evaluación he comprovado que tengo unos conocimientos competentes. Sin embargo creo que aún puedo aprender y formarme mucho más sobre estos temas. En cuanto a la justicia social creo que apenas tengo un nivel intuitivo y basado en el sentido común pero con una formación casi nula del tema. El campo del que tengo más conocimiento por mis estudios universitarios es sin duda el de la colaboración y trabajo en equipo a la hora de desarrollar un proyecto en grupo.

En conclusión, creo que en la universidad se debería hacer hincapié en el im-

pacto medioambiental que tienen los proyectos y dar más conocimientos sobre la gestión económica del mismo. Por lo demás creo que se da una buena docencia en el resto de los campos.

3.2 Presupuesto del proyecto

Para la elaboración de este proyecto y partiendo de los recursos que se han mencionado en apartados anteriores, se hará una estimación del coste del proyecto. Esta estimación se hará teniendo en cuenta los recursos de hardware, software, humanos y sus correspondientes amortizaciones. Además, se tendrán en cuenta los costes indirectos e imprevistos del proyecto.

3.2.1 Costes humanos

Este proyecto será realizado por una sola persona, por lo tanto los costes serán calculados mediante el número total de horas multiplicado por el precio por hora.

Tarea	**Duración (h)**	**Dedicación (h)**		
		Jefe de proyecto	**Desarrolador de software**	**Beta tester**
Realización de la fase inicial	90	30	30	30
Aprender sobre aprendizaje por refuerzo	100	20	60	20
Modificar Shattered Pixel Dungeon	50	5	35	10
Instalación y configuración del entorno	10	0	5	5
Buscar mejor técnica	25	5	10	10
Implementación, ejecución y testing de los agentes	170	10	100	60
Análisis y conclusiones	25	15	10	0
Redacción de la memoria anexo y documentación	40	20	20	0
Total	510	105	270	135

Cuadro 3.1: Tiempos por rol estimados.

3.2.2 Presupuesto de hardware

A continuación, se detallan los costes del hardware utilizado en este proyecto, así como la amortización estimada para cada uno

Rol	Horas	€/hora	Salario
Jefe de proyecto	105	50€/h	5.250 €
Desarrollador de software	270	40€/h	10.800 €
Beta tester	135	30€/h	4.050 €
Total	510		20.100 €

Cuadro 3.2: Coste estimado de los recursos humanos

Producto	Precio	Unidades	Vida útil	Amortización
PC	1.250€	1	4 años	75€
Monitor Samsung	150€	1	6 años	5€
Total	1.400€			80€

Cuadro 3.3: Coste estimado del hardware

3.2.3 Presupuesto software

Producto	Precio	Unidades	Vida útil	Amortización
Pycharm	0€	1	-	0€
Shattered Pixel Dungeon	0€	1	-	0€
Github	0€	1	-	0€
Microsoft Project 2016	0€	1	-	0€
Trello	0€	1	-	0€
Overleaf v2	0€	1	-	0€
SerpentAI	0€	1	-	0€
Nvidia CUDA	0€	1	-	0€
Total	0€			

Cuadro 3.4: Coste estimado del software

El coste de los recursos software es 0 ya que se utilizan todas las licencias gratuitas y software open source posibles. Shattered Pixel Dungeon, Github, Trello, SerpentAI y Nvidia CUDA son software Open Source gratuito. Las licencias de Pycharm y Microsoft Project 2016 se han conseguido gracias a ser estudiante de la UPC.

3.2.4 Costes imprevistos

Es difícil tener una estimación precisa de los costes en este tipo de proyectos. Existen muchos factores que pueden hacer que estos se alteren. A continuación se tendrán en cuenta los tres más sustanciales:

- **Aumento en el número de horas necesario:** En este tipo de proyectos de investigación sobre Machine Learning se necesitan una cantidad de horas muy elevada, más si la poténcia de computo no es muy elevada como es el caso. Sobretodo, en el entrenamiento de agentes es en donde más posibles retrasos puede haber y por lo tanto donde habría un augmento significativo del coste en horas.
- **Necesidad del uso de más hardware:** En el caso en que se necesite usar computación en la nube, los costes de hardware aumentarían.
- **Avería de los recursos hardware:** Si hubiese algún imprevisto con el hardware utilizado habría que remplazarlo de inmediato para poder seguir con el plan.

En el primer punto se dará un margen de 50 horas que equivalen a 2000 euros, en el segundo punto se dará un margen de 200 euros y en el tercer punto un margen de 300 euros. En total los costes inesperados son de 2500 euros.

3.2.5 Costes indirectos

A continuación, se mostrará la estimación de los costes indirectos para realizar el proyecto. Los costes de espacio son nulos debido a que el proyecto se realizará en un domicilio doméstico.

Producto	**Precio**	**Unidades**	**Coste estimado**
Electricidad	0.125€/kWh	1500kWh	188€
Internet	60€/mes	4 meses	240€
Espacio	0€	-	0€
Transporte	200€/ 4 meses	4 meses	200€
Total			628€

Cuadro 3.5: Costes indirectos

3.2.6 Costes totales

En la tabla siguiente se muestran los costes totales calculados a partir de los costes explicados anteriormente y aplicando además un 10 % de contingencia.

Concepto	Coste estimado
Costes de hardware	80 €
Costes de software	0 €
Costes humanos	20.100 €
Costes imprevistos	2.500 €
Costes indirectos	628 €
Total	23.308 €
Total + Contingencia(10 %)	25.638,80 €

Cuadro 3.6: Costes totales

3.2.7 Control de gestión de costes

El principal problema que puede surgir y que puede afectar al presupuesto final es la desviación temporal en alguna de las tareas del proyecto, sobretodo en la parte de entrenamiento de agentes, de la que se requiere una gran cantidad de tiempo.

Para controlar el presupuesto, al final de cada tarea, el pressupuesto se actualizará con la cantidad efectiva en horas, el coste de los recursos utilizados y los gastos de los acontecimientos inesperados que puedan ocurrir. Estas cifras se compararán con las estimaciones anteriores para obtener indicadores que mostrarán la cantidad de desviación de la planificación del presupuesto inicial. Se aplicarán las siguientes fórmulas:

$$\text{Desviación coste} = (CE - CR) \cdot HR$$

$$\text{Desviación consumo} = (HE - HR) \cdot CE$$

donde:

$$HE = \text{horas estimadas}$$

$$HR = \text{horas reales}$$

$$CE = \text{coste estimado}$$

$$CR = \text{coste real}$$

Debido a que el presupuesto se actualizará al final de cada tarea, esto ayudará a determinar donde se han producido, en caso de que sea necesario. Además, debido que se realizará un seguimiento de la cantidad de horas dedicadas a cada tarea y de los costes de los recursos utilizados en cada tarea, esto ayudará a determinar si la desviación se debe a una variación en el coste o en el consumo.

Además, se permite cierto margen de desviación ya que el coste de acontecimientos inesperados se ha tenido en cuenta y se ha aplicado un porcentage de contingencia a la estimación del presupuesto final.

3.3 Sostenibilidad e impacto social

Debido al carácter de investigación del proyecto, es complicado realizar un análisis en condiciones de la sostenibilidad, ya que lo que se esta realizando es una investigación sobre inteligencia artificial y Machine Learning. A continuación se describe la matriz de sostenibilidad y una explicación sobre sus filas.

	PPP	**Vida útil**	**Riesgos**
Ambiental	0.75kWh/0 emisión CO2	0.75kWh/0 emisión CO2	Más tiempo de entrenamiento/ gasto kW
Económico	25.638,80 €	Nuevas técnicas de IA	Ninguna
Social	Satisfactorio	Avance en IA	Otras técnicas que dejasen obsoletas las del proyecto

Cuadro 3.7: Matriz de sostenibilidad

3.3.1 Dimensión ambiental

El impacto ambiental de este proyecto es mínimo, el único recurso utilizado en este proyecto que tiene impacto es la corriente eléctrica utilizada para mantener el hardware encendido. Debido a esto, un gran análisis en el impacto medioambiental no tendría mucho sentido.De todas maneras se contestarán las preguntas que tengan sentido. El impacto ambiental de este proyecto es de unos 0.75kWh y 0 emisiones de CO2 ya que, solo se utiliza un PC para su desarrollo. Minimizar los recursos o reaprovecharlos no tiene sentido debido a que ya se trabaja con los recursos mínimos. Ambientalmente mi solución al problema no marcará la diferencia debido a que para entrenar a un agente se necesita mucho tiempo de cálculo sea la técnica que sea.

3.3.2 Dimensión económica

En cuanto a la dimensión económica, se ha llevado a cabo un estudio detallado de los costes implicados en este proyecto. El coste total estimado para este proyecto es de 27.090,80€, para llegar a esta cifra se han sumado los costes de hardware, software, humanos, imprevistos e indirectos (anteriormente detallados en este documento). Actualmente se paga de manera similar a los grandes investigadores de IA del mundo por lo que se ha tomado referencia de ellos. Este proyecto no dará claves para que el desarrollo de agentes tenga un coste menor sino que se aborda

un tema sobre los compañeros en los videojuegos que no se contemplaba anteriormente. Además, este proyecto será más económico que los proyectos en los que se basa(como OpenAI o AlphaGO) debido a que esos proyectos tienen muchos más recursos. También es cierto que cuantos más recursos más impacto medioambiental habrá.

3.3.3 Dimensión social

Como se decía en apartados anteriores, los beneficiarios de este proyecto serían las empresas de videojuegos que quisiesen aplicar las técnicas aquí descritas además de otros investigadores de inteligencia artificial y machine learning. A nivel personal, el proyecto es una manera de profundizar mis conocimientos en el mundo de la inteligencia artificial y en especial del reinforcement learning y puede servirme para futuros proyectos personales. Además, puede determinar si este será un campo al que me gustará dedicarme en un futuro. Este proyecto tiene como finalidad mejorar los agentes en videojuegos y darles un enfoque desde el punto de viste de un compañero, es necesario para que el campo de la inteligencia artificial siga adelante.

3.4 Leyes y regulaciones

En este proyecto no se involucran usuarios, clientes u otro tipos de personas y tanto el juego como todo el software usados en el mismo son libres y por lo tanto no tienen ninguna ley o regulación que les afecte.

Lo único que se puede destacar es la ley que se aplica a los videojuegos y los califica por edad, el PEGI. Debido a que Pixel Dungeon es un juego open source creado por un solo usuario no esta clasificado en el PEGI, por lo tanto se ha escogido un juego similar del mismo género para tener una idea. The binding of isaac es un juego muy similar que esta clasificado con PEGI 16, debido a las escenas de violencia y de drogas.

Capítulo 4

Pixel Dungeon: Modificación del juego y preproceso de píxeles

Antes de comenzar a implementar y entrenar a los agentes, para probar las técnicas de aprendizaje por refuerzo, se necesitaba implementar la modificación del juego y el preproceso para recoger información por pantalla. A continuación, se explicará todo el trabajo realizado.

4.1 Modificación del juego

El objetivo de la modificación era convertir el juego en cooperativo. Cuando se habló con el tutor sobre este trabajo se pensó en usar un juego *roguelike* como Pixel Dungeon, debido a que sus características eran muy curiosas para experimentar con los agentes, pero los juegos de este género acostumbran a ser de un solo jugador. Tras mucho indagar entre decenas de juegos se decidió modificar Pixel Dungeon, añadiendo un segundo jugador. Para la modificación del juego, se ha partido del código de este repositorio [15]. Se ha escogido partir desde aquí, debido a que es una versión que soporta gtx, y por lo tanto al compilar se generan archivos .jar que permiten ejecutar el juego en PC (originalmente era un juego de móvil).

4.1.1 Creación del compañero

Para crear al compañero se ha cogido la clase *Hero* y se ha instanciado un nuevo héroe con los parámetros adecuados en la escena. Este segundo héroe puede moverse ortogonalmente, al igual que el jugador principal y para ello se pueden escoger en el menú del juego qué teclas se quieren asignar para el comportamiento del compañero ej. Moverse una casilla hacia arriba. Además, el compañero también es capaz de interactuar con la celda en la que se encuentra, de modo que si es un

objeto, lo recoge, y si es una escalera, se sube o baja de piso (ver escenarios). Por último, el compañero es capaz de atacar a los enemigos que estén a distancia 1 ortogonalmente, si se mueve en la dirección en la que se encuentra dicho enemigo. Para poder distinguir bien a los dos personajes se ha fijado al compañero con la *skin* del guerrero, y cada vez que se empieza una partida se escoge al mago para la clase principal.

Al principio se pensó en que los dos jugadores tuvieran vidas distintas. Pero tras varias pruebas, se decidió que lo mejor, tanto visualmente, como para los agentes, era que los dos jugadores compañeros compartieran la misma barra de vida, de esta manera, cuando ataquen a uno de ellos, ambos perderán vida. Para regular este hecho se ha duplicado la vida inicial con la que se empieza en el juego, y también, se ha aumentado ligeramente la vida por nivel, que se gana al subir de nivel. Ambos jugadores ganan experiencia por las acciones que hacen los dos y por lo tanto ganan la misma cantidad de experiencia y suben de nivel a la vez.

4.1.2 Microturnos y nuevo gameplay

Es importante recalcar que Pixel Dungeon es un juego por microturnos, y este es un factor muy importante a tener en cuenta. En el juego original los enemigos se mueven en el momento en que el personaje principal realiza algún movimiento o ataca. Además, si este se intenta mover contra un muro, quedándose en el mismo lugar, el juego no lo contará como acción y los enemigos no se moverán. Esto ha sido uno de los aspectos más complejos para implementar, debido a que es un sistema pensado para solo un jugador, y se quería modificar para incluir al compañero en esta transición de microturnos. Por ello, lo que se ha hecho es mantener el hecho de que los enemigos ataquen solo cuando el jugador principal se mueva y que el compañero haga su movimiento después. De manera que el orden de los turnos para los agentes es: jugador principal - enemigos - compañero. Esto generará algunas posibles estrategias que pueden romper el juego, y que alguno de los agentes usarán.

4.1.3 Autoguardados

El juego original constaba de unos autoguardados a cada acción que se realizaba, de esta manera, aunque cerraras el juego en mitad de una pelea, al volverlo a abrir seguirías en un punto igual o muy similar. Para el entrenamiento de los agentes era interesante que el punto de partida fuese siempre el mismo al reiniciar una partida guardada. Por lo tanto, lo que se ha hecho es fijar los archivos de guardados en el punto que interesaba para el escenario y desactivar el autoguardado. De esta manera, siempre que se cargue la partida se empezará desde el mismo punto del juego.

4.1.4 Problemas encontrados

Los principales problemas encontrados al modificar el juego fueron entender el funcionamiento del código asociado al héroe, y poder crear una nueva instancia del mismo sin alterar ninguna función vital del juego. Además, como se explica anteriormente, se tuvieron problemas al cuadrar los microturnos del compañero con el flujo del juego, y hubieron varios errores al implementar la interacción con la celda actual que hacían que el juego se cerrara.

4.2 Descripción del escenario

4.2.1 Escenario inicial

Inicialmente, el escenario de juego de los agentes consistía en jugar al juego de forma "habitual". Los agentes empezaban al inicio del juego y su objetivo era descender el máximo número de pisos posibles. Cada vez que se moría, se creaba una nueva partida con la que se generaba un nuevo mundo aleatorio. Esto era un problema, ya que retrasaba el aprendizaje del agente. Además, en este punto del proyecto solo se utilizaba la información que se podía obtener de los píxeles, por lo que era difícil conseguir poner valor a las recompensas que otorgaban ciertas acciones.

Figura 4.1: Reaparición de los agentes en el escenario inicial.

Con este escenario se hicieron múltiples pruebas con el algoritmo DDQN, hasta al final decidir substituirlo por el escenario final escogido. El problema que tenia este escenario es que la magnitud de posibilidades que podían hacer los agentes, el entorno, era inmenso. Cada vez que los agentes morían se generaba una nueva partida con un nuevo mundo aleatorio. Además, esto hacía mucho más difícil la tarea de escoger los valores de las recompensas, hasta tal punto, que se necesitaba un tiempo de entrenamiento inviable para que los agentes hicieran algo productivo. Por ese mismo motivo, se desecho este escenario y se empezó a trabajar en el definitivo.

4.2.2 Escenario final

Mientras se estaba probando el primer escenario propuesto y al ver todos los problemas que tenía, se me ocurrió, que una manera de simplificar el escenario sería dar un entorno fijo que no cambie y que tenga mucha acción para los agentes. Este entorno es: una batalla contra un jefe final. En el juego, cada cinco pisos hay un piso especial con un monstruo difícil de vencer, no se puede avanzar hacia el siguiente piso hasta haber derrotado a dicho enemigo. Para ello, se modificó el juego para que dejase de guardar partida, de esta manera cada vez que se volvía a cargar la partida se empezaba desde el mismo punto.

Figura 4.2: Reaparición de los agentes en el escenario final.

El escenario final consiste en los dos agentes reapareciendo a distancia 2 y 3 del jefe final, el personaje principal está equipado con las armas que se han recogido

a lo largo de los cinco pisos (objetos que facilitan más el vencimiento de este enemigo), pero en cambio, el compañero tiene las estadísticas base sin suplementos por objetos. Ambos están a nivel 5, el nivel recomendado para este piso que les otorga más fuerza y vida. La barra de vida del enemigo se ve en la parte superior de la pantalla, justo debajo de la vida de los agentes. Este escenario es vencido cuando los agentes matan al jefe y son derrotados cuando mueren a manos del jefe. No hay ningún otro desenlace posible, es morir o matar.

Goo, la ballena

El enemigo de los agentes es Goo, una ballena que hace de primer jefe final del juego. El enemigo tiene un patrón simple, persigue a los agentes y los ataca, además tiene un ataque en área que si coge a los dos agentes a la vez, les quitará el doble de vida. Cuando Goo tiene poca vitalidad entra en modo *bersek* haciendo más daño con sus ataques y aguantando más los golpes.

Goo dispone de un ataque cargado en área que empieza a crear partículas a su alrededor y al cabo de unos cuantos turnos libera la energía infligiendo mucho daño. Durante el lanzamiento de este ataque Goo permanece inmóvil.

Un punto muy importante a tener en cuanta sobre Goo, es que al ser un jefe final tiene una gran cantidad de esquiva. Eso significa que algunos de los golpes que le dan los agentes pueden ser esquivados evitando el daño por completo. Es un factor importante a tener en cuenta debido a que por culpa del azar, en el entrenamiento los agentes pueden ser despistados al golpear y no tener un efecto positivo.

4.3 Técnicas de visión por computador: Pixels to Information

Se han aplicado algunas técnicas de visión por computador para leer la información en la pantalla (píxeles) y así, poder decidir cuándo dar recompensas negativas o positivas, dependiendo de lo que esté pasando en el estado del juego actual.

4.3.1 Puntos de vida

Para detectar la vida que se tiene en cada momento, se observa el subconjunto de píxeles que pertenece a la barra de vida, se parte la barra de vida en cinco partes y se compara el nivel de rojo (RGB) de cada una de las partes. Puesto que, cuando la barra está vacía se vuelve completamente gris, se puede saber fácilmente si un trozo de barra está completamente llena, parcialmente llena o completamente

vacía. Si una barra está completamente llena se dan 10 puntos de vida (HP) si está parcialmente llena se otorgan 5 puntos de vida y si esta vacía se otorgan 0 puntos de vida. En total, el jugador dispone de 100 puntos de vida. Una pequeña optimización que se ha hecho es dejar de calcular en cuanto se vea la primera barra medio vacía. Además, debido a problemas cuando el personaje tenía muy poca vida y apenas había un píxel de la barra de vida en rojo, cada vez que se detectaba que se tenía cero de vida, se hacia una comprobación de los píxeles que daban problemas para comprobar si realmente el agente estaba muerto.

Este sistema se ha usado, tanto para la vida de los agentes, como para comprobar la vida del jefe final Goo.

4.3.2 Distancia entre héroes

Tras varias pruebas solamente con la información de los píxeles, y tras hablar con el tutor, se decidió usar información del juego (variables internas). Para ello, debido a que, como he explicado anteriormente, el juego se ejecuta desde un *.jar* o un *.exe*, de donde es muy complicado extraer el valor de las variables, se ha usado un fichero de texto auxiliar donde se escribe en cada iteración del juego la información deseada. En este caso, la información más útil para los agentes es la distancia que hay entre ellos. Esta información se usará para los estados y para calcular la recompensa. La posición de cada agente está definida por un entero y la distancia entre ellos se calcula mediante la anchura del mapa. Supongamos que tenemos dos enteros, a y b que son las posiciones del personaje principal y el compañero, entonces para calcular las coordenadas correspondientes haríamos:

$$ax = a\,\%width \tag{4.1}$$

$$ay = a/width \tag{4.2}$$

Se hace exactamente lo mismo con el punto b, y para calcular la distancia se retorna el máximo de la resta de coordenadas en valor absoluto.

$$Distance(a, b) = max(abs(ax - bx), abs(ay - by)) \tag{4.3}$$

Al ser un juego que consume muy pocos recursos, hacer este cálculo por cada iteración en el juego es posible sin ralentizarlo de ninguna manera, incluso escribiendo la distancia en un archivo de texto.

Capítulo 5

Aprendizaje por refuerzo

El aprendizaje por refuerzo [12] hace referencia a los algoritmos *goal-oriented*, que aprenden como alcanzar un objetivo complejo o maximizar la probabilidad de cumplir el mismo mediante muchos pasos; por ejemplo, maximizar los puntos que se obtienen en un juego después de muchos movimientos. Estos algoritmos pueden empezar como una hoja en blanco y con las condiciones adecuadas alcanzar comportamiento superhumano. Como un niño incentivado por caramelos, estos algoritmos son penalizados cuando toman malas decisiones y recompensados cuando realizan las correctas - esto es conocido como refuerzo.

El aprendizaje por refuerzo resuelve el difícil problema de la correlación inmediata entre acciones y las consecuencias retardadas que provocan las mismas. Como los seres humanos, los algoritmos que usan aprendizaje por refuerzo a veces tienen que esperar un tiempo para ver el fruto de sus decisiones. Operan en un entorno retardado, donde puede ser difícil entender que acción lleva a que desenlace a través de muchos instantes de tiempo.

A diferencia de otros enfoques (como aprendizaje supervisado), en el aprendizaje por refuerzo no se cuenta con datos o ejemplos de comportamientos o acciones ejemplares. El entrenamiento se realiza a traves de la interacción del agente con el entorno.

El aprendizaje por refuerzo es una de las tres grandes ramas del aprendizaje automático junto al aprendizaje supervisado y el aprendizaje no supervisado.

5.1 Conceptos básicos

El aprendizaje por refuerzo puede ser entendido usando el concepto de agente, entorno, estados, acciones y recompensas. Definamos los conceptos básicos del aprendizaje por refuerzo:

- **Agente:** Un agente realiza acciones; por ejemplo, un dron lleva una mercancia, o en Super Mario navegando en el juego. El algoritmo es el agente. En la vida real, el agente eres tu.
- **Entorno:** El mundo por el que el agente se mueve. El entorno recoge el estado actual del agente y la acción realizada como entrada y retorna como salida la recompensa del agente y su siguiente estado. Si tu eres un agente, el entorno podrían ser las leyes de la física o las reglas de la sociedad que determinan las consecuencias de una acción.
- **Estado (s):** Un estado es una situación concreta donde se encuentra el agente; por ejemplo un momento específico, una configuración que pone al agente en relación con otra cosa como por ejemplo una herramienta o un obstáculo. Es la situación en la que se encuentra el agente en un instante de tiempo.
- **Acción (a):** Una acción es la maniobra, actividad o movimiento que el agente escoge realizar en un instante de tiempo determinado. Hay que remarcar que el agente escoge de una lista de posibles acciones. En los videojuegos, la lista puede incluir ir a la izquierda o a la derecha, saltar o deslizarse.
- **Recompensa (r):** Una recompensa es el *feedback* con el que se mide el éxito o fracaso de la acción de un agente. Por ejemplo, en un videojuego, cuando Mario toca una moneda gana puntos. Desde cualquier estado, un agente envia una salida en forma de acciones al entorno y este retorna el nuevo estado del agente (resultante de la acción realizada desde el estado anterior) a la vez que una recompensa. Las recompensas pueden ser inmediatas o retardadas. Evaluan las acciones de un agente de manera efectiva.
- **Política (π):** Una política es la estrategia que el agente utiliza para determinar la siguiente acción del agente basandose en el estado actual. Mapea estados con acciones, las acciones que pronostican las recompensas más elevadas.

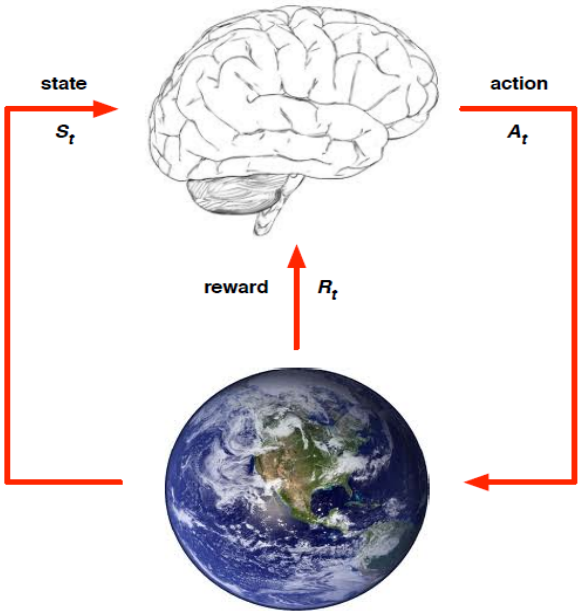


Figura 5.1: Esquema de un agente de aprendizaje por refuerzo (imagen extraida de las diapositivas de David Silver, Deepmind).

Las interacciones del agente con el entorno ocurren en tiempo discreto (en pasos) y se modelan como un problema de decisión de Markov(MDP).

Un problema de decisión de Markov está definido por:

- Un conjunto de estados. Los estados son una función de la información del agente acerca del entorno.
- Un conjunto de acciones posibles en cada estado. Las acciones representan las diferentes maneras en las que un agente puede interactuar con el entorno.
- La probabilidad de transición de un estado **s** a otro estado **s'** dada una acción.
- La función de recompensa en transición de **s** a **s'** dada una acción.

Estas dos últimas definen el modelo del mundo. Si este es conocido, tan solo hay que calcular directamente la estrategia óptima mediante enfoques como la programación dinámica. Si el modelo es, por el contrario, desconocido (como en este caso), hay que aproximar aprendiendo estimaciones de recompensas futuras obtenidas al elegir una acción a en un estado s.

5.2 Modelos de aprendizaje por refuerzo

Dentro de la enorme rama del aprendizaje por refuerzo encontramos distintos tipos de modelos y formas de afrontar la recopilación de información en el entrenamiento. A continuación se explicán las diferencias entre los distintos tipos de modelo que existen en el aprendizaje por refuerzo.

5.2.1 Exploración vs Explotación

El aprendizaje por refuerzo se basa en realizar decisiones que llevan a dos caminos fundamentales:

- *Explotación*: Tomar la mejor decisión dada la información actual
- *Exploración*: Recolectar más información

La mejor estrategia a largo plazo puede implicar sacrificios a corto plazo y por otro lado recolectar información suficiente para hacer las mejores decisiones es algo de vital importancia.

Escoger qué variables aleatorias son las mejores requiere saber las distribuciones de probabilidad reales y ya que las distribuciones de probabilidad empiricas son aproximaciones, si un agente no explora lo suficiente puede darse el caso de tener un coste por oportunidad perdida por no escoger la mejor elección posible en ese momento. Por otro lado si el agente sólo explora entonces tendra un coste por oportunidad perdida cada vez que la exploración sea una no óptima. Por ello se debe conseguir un balance entre probar cosas nuevas y escoger la mejor opción actual.

La solución para conseguir este balance depende de las restricciones de las distribuciones de recompensa y de cada cuanto el agente puede escoger una nueva acción. Para ello se usa ϵ-greedy (epsilon greedy), el parametro ϵ está entre 0 y 1. El agente escogerá la acción que de mejores resultados (explotación) con una probabilidad de 1 - ϵ y en otro caso escogerá una acción aleatoria (exploración). Si el valor de ϵ es fijo entonces suponiendo que fuese 0.5 el agente escogeria la acción óptima el 50 por ciento de las veces. Por ello para evitar esto lo que se hace es empezar por un valor de ϵ alto y se va disminuyendo a lo largo del tiempo, de esta manera al principio se harán muchos movimientos de exploración y al final muchos más de explotación pero habiendo explorado previamente lo suficiente.

5.2.2 On policy vs Off policy

Un agente *on-policy* aprende el valor basandose en su acción actual derivada de su actual política mientras que los agentes *off-policy* como su contraparte aprenden basandose en la acción a obtenida de otra política. En Q-learning por ejemplo esta política es la *greedy policy*. Por lo tanto un algoritmo *on-policy* como por ejemplo Sarsa aproxima su política dependiendo de las acciones del agente.

5.2.3 Algoritmos

Hay tres clases de algoritmos de aprendizaje por refuerzo:

- **Programación dinámica**: Esta clase de algoritmos se usan para computar las políticas óptimas dado un modelo perfecto del entorno como un Proceso de decisión de Markov(MDP). Los algoritmos clasicos de programación dinámica son de poca utilidad debido a que se asume un modelo perfecto y por su gran coste computacional, aun así siguen siendo muy importantes teóricamente. La idea principal de la programación dinámica y de el aprendizaje por refuerzo en general, es el uso de las funciones de valor para organizar y estructurar la búsqueda de buenas políticas.
- **Métodos de Monte Carlo**: Los métodos de Monte Carlo sólo requieren de una muestra de experiencia, una secuencia de estados, acciones y recompensas de una interacción real o simulada con un entorno. Pero el conocimiento del mundo no es necesario. Aunque se requiere de un modelo, el modelo necesita solo generar las muestras de algunas transiciones y no las distribuciones de probabilidad completas de todas las transiciones posibles que son necesarias en programación dinámica.
- **TD Learning**: Si se tuviera que identificar una idea como la principal para el aprendizaje por refuerzo esta seria sin duda *temporal difference learning*. Esta es una combinación de las ideas de Monte Carlo y de programación dinámica. Como en los métodos de Monte Carlo, los métodos *TD* pueden aprender directamente de experiencia sin necesidad de un modelo del entorno dinámico. Y como en programación dinámica, los métodos *TD* estiman basandose en parte en otras estimaciones sin esperar al resultado final. Utilizan *bootstrapping*.

En este proyecto se usarán métodos de *TD Learning*.

Capítulo 6

Descripción del agente

6.1 Objetivos

El objetivo principal del agente que controla al compañero es ayudar a vencer al jefe enemigo Goo, en el menor tiempo posible, con la menor cantidad de vida perdida posible. Para ello se han entrenado a dos agentes a la vez debido a la imposibilidad de poder jugar controlando al personaje principal la gran cantidad de horas de entrenamiento que se requiere en este tipo de juegos. Lo que se ha hecho es entrenar dos agentes a la vez intentando maximizar su cooperación.

6.2 Definición de estados y acciones

Para los primeros dos algoritmos Q-lerning y SARSA, se han realizado 3 representaciones distintas: por vida, por diferencia de vida y por distancia. En los dos últimos algoritmos se han hecho representaciones por imagenes de los píxeles de la pantalla. En cada uno de los algoritmos se explicará con más detalle su representación de estados.

En cuanto a las acciones posibles hay 9 de ellas para cada uno de los personajes: Moverse ortogonalmente (arriba, abajo, izquierda, derecha, norteste, sudeste, noroeste y noreste) y la acción de atacar. Las acciones se han mantenido igual para todos los algoritmos y modelos.

Se decidió no utilizar los objetos utilizables como la varita debido a que el compañero no era capaz de usarlos.

6.3 Recompensas

La función recompensa varia segun los modelos. Se explicarán todos los detalles más adelante en cada uno de los algoritmos.

6.4 Entrenamiento y resultados

Debemos tener en cuenta que para entrenar al agente, se necesita una gran cantidad de horas y cuando se hace, el juego debe estar ejecutado en primer plano y el script es el encargado de jugar presionando las teclas pertinentes y moviendo el ratón. Por este motivo, el ordenador donde se entrena queda inutilizado durante las horas de entrenamiento, y puesto que solo se dispone de un ordenador capaz de entrenar, la gran mayoría de los modelos han sido entrenados durante la noche. Por este motivo los tiempos de entrenamiento de los distintos modelos son variables y algunos modelos han estado mucho más tiempo entrenando que otros por las circustancias del momento cuando fueron entrenados.

La puntuación con la que se ha medido el éxito o fracaso del agente esta hecha después de hacer 100 pruebas con cada modelo. A partir de ello se obtienen dos puntuaciones: una puntuación de muertes, siendo el número de veces que ha vencido al jefe final y otra puntuación de vida donde se mide la cantidad de vida media que se ha inflingido al jefe final. De esta manera la máxima puntuación tanto de muertes como de vida es de 100.

A continuación se describirán todos los algoritmos que se han utlizado en este proyecto y los diferentes modelos entrenados con sus respectivos resultados mediante la puntuación anteriormente explicada.

Capítulo 7

Q-Learning

7.1 Descripción y funcionamiento

Q-learning [5] es un algoritmo *off-policy* de la clase TD-learning que se basa en los valores Q(s,a). En Q-learning definimos la función Q(s,a), que representa la máxima recompensa futura aplicando el descuento cuando realizamos una acción a en un estado s y continua de manera óptima desde este punto en adelante.

$$Q(S_t, A_t) = maxA_{t+1} \tag{7.1}$$

La manera de entender Q(s,a) es como "la mejor puntuación al final de cada turno después de realizar la acción a en el estado s"'. Es llamado *Q-function* porque representa la calidad(quality) de una acción dada en un estado dado.

Para calcular *Q-function* debemos centrarnos en una sola transición s, a, r, s' . Podemos expresar el *Q-value* de un estado s y de una acción a en términos de el *Q-value* del estado siguiente *s'*.

$$Q(s, a) = r + \gamma maxQ(s', a) \tag{7.2}$$

Esta equación es la famosa equación de Bellman. El máximo de la recompensa futura para un estado y una acción determinados, es la recompensa inmediata sumada al máximo de la recompensa futura en el siguiente estado.

La idea principal en *Q-learning* es que podemos aproximar *Q-function* de manera iterativa usando la equación de Bellman. En el caso más simple *Q-function* es representado como una tabla donde las filas son los estados y las columnas son las acciones. En el caso de este proyecto se ha representado Q-function como un diccionario (acción, estado) : valor. A continuación se muetra el pseudocódigo del algoritmo.

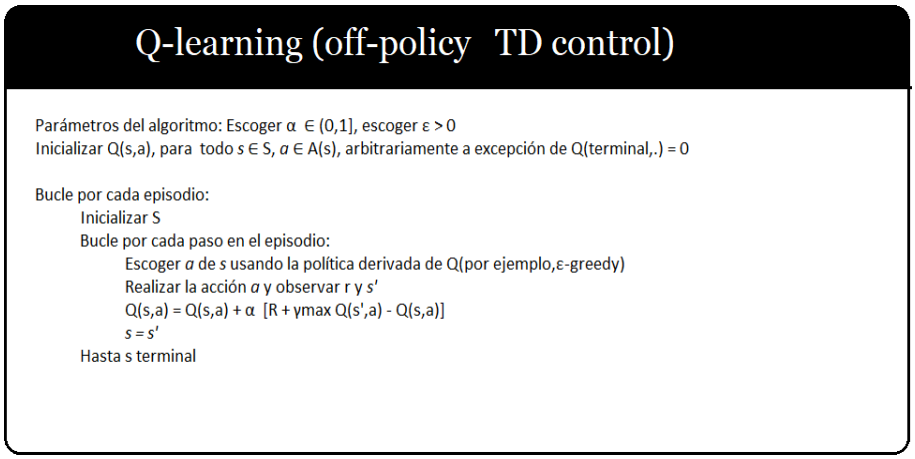


Figura 7.1: Pseudocódigo de Q-learning.

Inicialmente la tabla se inicializa de manera arbitraria. Cada vez que el agente selecciona una acción a en un estado s se le retorna un estado s' y una recompensa inmediata r y con esta nueva información se actualizan los *Q-values* a partir de la equación de Bellman explicada anteriormente. En el algoritmo α es el ratio de aprendizaje que determina hasta que punto la nueva estimación de Q sobrescribirá la anterior. Además, λ es el factor de descuento que determina la importancia de las recompensas recientes respecto a las recompensas posteriores en el tiempo. Cuanto más cerca de 1 más importancia se le da a las recompensas futuras. Ambas constantes están comprendidas entre 0 y 1.

7.2 Representación de estados y recompensas

Para Q-learning se han probado diversos estados y recompensas. Se probaron tres tipos de representaciones de estados distintas para ver cual era mejor y valorar la importancia de una buena representación.

Primero se probo con el estado siendo la resta de la vida actual de los protagonistas y la vida actual del jefe final. De esta manera, los estados eran números enteros múltiples de 5 entre -100 (si el héroe muere y el jefe final tiene toda la vida) y 100 (si el héroe mata al jefe final sin recibir ningún golpe). La segunda forma de representar los estados que se probó fue con la vida actual del jugador siendo un entero entre 0 y 100. Por último, la se probó una representación que usará información fuera de los píxeles, la distancia entre el protagonista y el compañero. Se hara una comparación sobre las distintas representaciones de estados en los distintos algoritmos en la sección 11.

En cuanto a las recompensas, primeramente se hizo una primera versión muy simple, donde se daba 1 punto si dañabas al enemigo y -0.05 si no le dañabas.

Luego se daba -1 punto si el enemigo inflingía daño a los agentes y 0 puntos si no lo hacía.

Más adelante, se probó a dar 0.1 puntos por cada punto de vida inflingido al enemigo, de esta manera, si de un golpe le quitas 10 puntos de vida, esa acción retorna 1 punto. Luego de la misma forma se quita 0.1 puntos por cada punto de vida perdido a manos del enemigo. Además, para impedir el problema de estancamiento en un muro (explicado en 12.1.1) se quitan 5 puntos de recompensa si se mantiene la misma posición y estado durante más de 15 turnos. Por último se intento añadir la distancia entre héroes, si la distancia entre héroes es menor a 5 (se asume que estan lo suficientemente cerca) se suman 0 puntos de recompensa, en otro caso si la acción realizada ha alejado a ambos héroes se restan 0.5 puntos de recompensa y si se acercan se dan 0.1 puntos. Después de probar las tres distintas formas de calcular la recompensa la que dió mejores resultados fue la segunda manera.

Result: Función recompensa definitiva para Q-learning
reward = 0 **if** *actualVida ¡anteriorVida* **then**
  reward += (diffencia/5)*-1;
else
  reward += 0.05;
end
if *actualVidaBoss ¡anteriorVidaBoss* **then**
  reward += (diffenciaBoss/5)*1;
else
  reward += -0.05;
end
if *estancado* **then**
  reward += -5;
else
  r
end
eturn reward;

Algorithm 1: Pseudocódigo función recompensa Q-learning

7.3 Entrenamiento y resultados

7.3.1 Modelo sin distancia y diferencia de vida como representación de estados

Para primer modelo, que no usa distancia y que usa como representación de estados la diferencia de vida entre los héroes y el enemigo, se dejó entrenando *21 horas, 10 minutos y 29 segundos*. Se producieron *1012* episodios en ese tiempo y un total de *76265* pasos sumando todos los episodios. En todos estos episodios se venció al

enemigo final *96* veces.

Usando el primer modelo de Q-learning el agente muere en la gran mayoría de los intentos, los agentes se separan demasiado cuando reciben daño y esto fácilita al jefe final el acabar con ellos.

Puntuación de muertes: 11

Puntuación de vida: 57

7.3.2 Modelo sin distancia y vida como representación de estados

Para entrenar el segundo modelo que usa como representación de estados la vida aliada, se ha dejado entrenando *13 horas, 31 minutos y 32 segundos*. Se producieron *3414* episodios y un total de *36994* pasos como suma de los episodios. En todos estos episodios el enemigo final es derrotado *22* veces.

Usando el segundo modelo de Q-learning el agente muere en la gran mayoría de los intentos, los agentes intentan huir del enemigo separandose y muriendo igualmente.

Puntuación de muertes: 8

Puntuación de vida: 41

7.3.3 Modelo con distancia como representación de estados

Para entrenar el tercer modelo que usa distancia como representación de estados, se dejó entrenando *12 horas,16 minutos y 57 segundos*. Se realizaron *3324* episodios con una suma total de *29979* pasos donde el enemigo final fue derrotado *8* veces. Aunque este modelo haya estado menos tiempo entrenando, podemos observar que ha muerto mucho más rápido y por ende tiene muchos más episodios que el primer módelo. Como ya anticipa el entrenamiento este tercer modelo es peor que el primero.

Usando el algoritmo de Q-learning con distancia el agente muere en la gran mayoría de los intentos, los agentes no colaboran prácticamente nada entre ellos y reciben mucho daño de los ataques en área del enemigo, esto puede darse porque la representación de estados no es la correcta.

Puntuación de muertes: 7

Puntuación de vida: 32

7.3.4 Explicación resultados

Aunque ninguno de los tres modelos es realmente bueno, se puede ver claramente que el primer modelo es el mejor de ellos ya que mata más al enemigo final y hace una media de daño superior al resto. Hay varios posibles motivos por lo que el agente no vaya tan bien como se esperaba: el algoritmo no es muy potente para escenarios muy complejos y una batalla contra un jefe final es de los escenarios más dificiles en un juego de estilo roguelike. Otro posible motivo es que la representación de estados sea incorrecta o que no se hayan dado suficientes estados aun habiendo probado varias asignaciones de estados. Por último puede que las recompensas hayan dado lugar a muchos falsos positivos debido a la problemática de chocarse contra la pared (ver sección 12.1.1).

Para entrenar se ha utilizado ϵ = variable de 0.999 hasta 0.01, $\alpha = 0.2$ y $\gamma = 0.9$, se han probado varios valores del factor de descuento pero al final el que daba mejores resultados es el mencionado.

Capítulo 8

SARSA

8.1 Descripción y funcionamiento

SARSA [5] es un algoritmo *on-policy* de la clase TD-learning y su nombre son las siglas de: state, action, reward, state y action. SARSA se asemeja mucho a Q-learning, la diferencia principal entre SARSA y Q-learning es que SARSA es un algoritmo *on-policy*. Esto implica que SARSA aprende el *Q-value* basandose en las acciones realizadas durante la política actual en vez de una política *greedy*.

El primer paso es aprender una función de acción-valor preferible a una función estado-valor. En particular, para un método *on-policy* tenemos que estimar q_π(s,a) para el comportamiento actual, la política π y todos los estados s y acciones a. Recordar que un episodio consite en alternar la sequencia de estados en pares estado-acción.

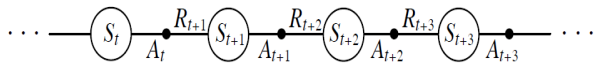


Figura 8.1: Secuencia de estados en SARSA.

Consideramos transiciones de pareja estado-acción a pareja estado-acción y aprendemos los valores de las parejas estado-acción. Son cadenas de Markov con procesos de recompensa. Los teoremas aseguran convergencia de un estado en TD(0) donde también se aplican en los algoritmos correspondientes para los valores de acción.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t] \tag{8.1}$$

Esta actualización se hace después de cada transición que viene de un estado

no terminal S_t. Si S_{t+1} es terminal, entonces Q(S_{t+1},A_{t+1}) se define como cero. Esta regla usa cada elemento de la tupla de cinco elementos de los eventos (S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}) que definen una transición de una pareja estado-acción a la siguiente.

Las propiedades de convergenia de SARSA dependen de la naturaleza de la dependecia de la política usadda con Q. SARSA converge con probabilidad 1 en una política y función acción-valor óptima, mientras que todos las parejas estado-acción sean visitados un número infinito de veces.

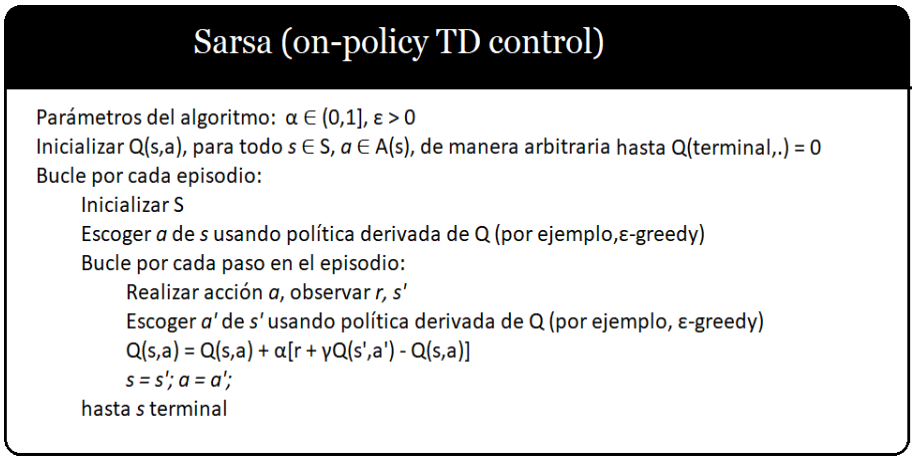


Figura 8.2: Pseudocódigo de Sarsa.

8.2 Representación de estados y recompensas

Al igual que con Q-learning se probaron diferentes representaciones de estados creando distintos modelos con: vida, diferencia de vida y distancia. Se usó la misma función de recompensa que en Q-learning.

8.3 Entrenamiento y resultados

Al igual que con Q-learning se han entrenado tres modelos distintos con distintas representaciones de estados.

8.3.1 Modelo sin distancia y diferencia de vida como representación de estados

Para entrenar el modelo sin distancia y con diferencia de vida como representación de estados, se dejó entrenando *9 horas, 4 minutos y 11 segundos*. En este tiempo transcurrieron *669* episodios con una suma total de *52429* pasos. Se derrotó al jefe final *86* veces.

Usando el algoritmo de SARSA este primer modelo puede reflejar que aunque la puntuación de vida es muy similar a la que daba Q-learning la puntuación de muerte es doblada. Aun así, los resultados son realmente malos y muy lejos de un comportamiento bueno al igual que pasaba con Q-learning.

Puntuación de muertes: 26

Puntuación de vida: 55

8.3.2 Modelo sin distancia y vida como representación de estados

Para entrenar este modelo se ha dejado entrenando *20 horas, 29 minutos y 5 segundos*. En ese tiempo se han producido *3920* episodios con una suma total de *77040* pasos. Se ha derrotado al jefe final *74* veces.

En este segundo modelo vemos como ambas puntuaciones disminuyen ligeramente como pasaba con Q-learning, esta representación de estados da peores resultados.

Puntuación de muertes: 20

Puntuación de vida: 51

8.3.3 Modelo con distancia como representación de estados

Para entrenar el modelo con distancia como representación de estados se dejó entrenando *1 día, 14 horas, 31 minutos y 29 segundos*. En este tiempo se realizaron *1730* episodios cuyos pasos sumados daban un total de *134226* pasos. El enemigo final fue derrotado 200 veces. Podemos observar que aunque este modelo estuvo entrenando más tiempo que los anteriores, el número de episodios no es muy elevado, esto se debe a que algunos episodios duraban horas debido al problema del movimiento contra un muro(explicado en la sección 12.1.1).

Con distancia, el agente muere en la mayoría de las veces también, pero igual

que en los otros modelos se mejora a Q-learning.

Puntuación de muertes: 23

Puntuación de vida: 54

8.3.4 Explicación resultados

Al igual que pasaba en Q-learning podemos observar que el mejor modelo es el primero, represenación de estados como diferencia de vida. Aunque el agente muera un 75 % de las veces, en SARSA podemos observar como todos sus modelos mantienen un daño al jefe final mayor a 50 puntos. Aunque los resultados sean malos podemos ver que en general son mejores que en Q-learning. Los motivos por los que los resultados no son buenos son los mismos que en Q-learning (ver sección 7.3.4).

Al igual que en Q-learning se ha utilizado ϵ = variable de 0.999 hasta 0.01, α = 0.2 y $\gamma = 0.9$, se han probado varios valores del factor de descuento pero al final el que daba mejores resultados es el mencionado.

Capítulo 9

Deep Q-Network

9.1 Descripción y explicación redes neuronales

9.1.1 Redes neuronales

Las redes neuronales [17] son una idea utilizada generalmente para machine learning. Estan basadas en como funciona un cerebro humano y su idea es simular múltiples neuronas interconectadas dentro de un ordenador de manera que sean capaces de aprender cosas, reconocer patrones y realizar decisiones de una manera *humana*. Lo increible sobre las redes neuronales es que no se tienen que programar el aprendizaje explícitamente, aprenden por si solas como si de un cerebro humano se tratase.

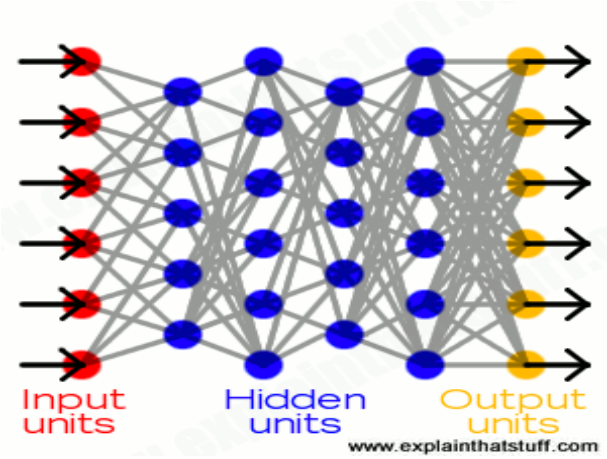


Figura 9.1: Ejemplo de red neuronal. Imagen extraida de [17]

Una red neuronal típica tiene desde unas pocas decenas hasta cientos, miles o millones de *neuronas* alineadas en una serie de *capas*, cada una de estas capas

conectada a las capas que tiene a sus dos lados. Algunas de ellas son *capas de entrada*, diseñadas para recibir varias formas de información del mundo exterior, la cual la red intentará aprender, reconocer o procesar. Otras son llamadas *capas de salida* y se encuentran en el lado opuesto a las capas de entrada, estas devuelven la respuesta de la red neuronal a la información que ha sido aprendida. El resto de capas que se encuentran entre las capas de entrada y las capas de salidas son llamadas *capas ocultas*. La mayoría de las redes neuronales son *fully-connected* lo que significa que cada neurona dentro de las capas ocultas esta conectada a toda neurona de las capas contiguas a ella. Las conexiones entre una neurona y otra estan representadas por un número llamado *peso* que puede ser positivo (si una neurona alienta a otra) o negativo (si una neurona suprime o inhibe otra). Cuanto más alto sea el valor del peso, más influencia tiene una neurona sobre otra. Este proceso se corresponde a como funcionan las neuronas reales de un cerebro humano y es llamado sinapsis.

La información fluye en la red neuronal de dos maneras distintas. Cuando está aprendiendo (entrenamiento) y cuando se ejecuta normalmente (después de entrenar). En general, la información va hacia delante, de izquierda(capas de entrada) a derecha(capas de salida) en un proceso llamado *feed-forward*. No todas las neuronas se procesan a la vez, cada neurona recibe valores de entrada de las neuronas de su derecha a las cuales esta conectada y multiplica estos valores por los pesos de las conexiones por las que viajan. Cada neurona suma todos los valores recibidos en esa dirección y se aplica una función de activación a la suma, que determina si la neurona activa su conexión con su neurona de la derecha y sigue propagándose o se apaga.

Hay distintos tipos de funciones de activación:

- **Threshold**: Si el valor que llega a la función sobrepasa un cierto límite la neurona se propaga y sino no. Esta función es binaria no da pie a activaciones parciales.
- **Sigmoide**: Es una función de la forma de:

$$f(x) = 1/1 + e^{-x} \tag{9.1}$$

  Su valor esta entre 0 y 1 por lo que es una función no binaria. Es sencilla de entender y de aplicar pero tiene grandes problemas como: el problema del gradiente, su salida esta entre 0 y 1 lo que hace que sea más difícil de optimizar, los sigmoides saturan los gradientes y convergen lentamente.

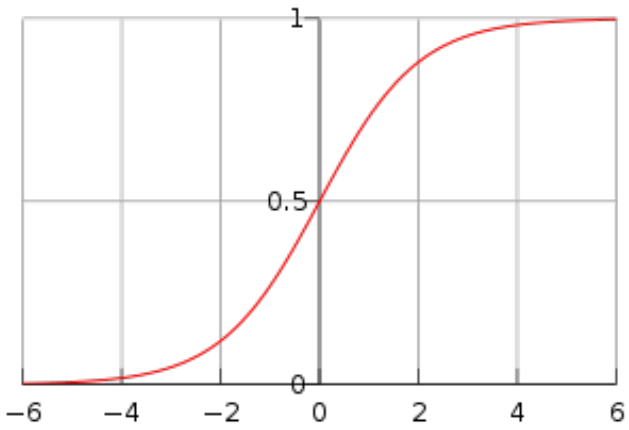


Figura 9.2: Función sigmoide.

- **ReLu**: La función ReLu (*rectified linear units*) es de la forma:

$$R(x) = max(0, x) \tag{9.2}$$

  El problema principal de ReLu es que en la zona de las X negativas, el gradiente será 0 y eso puede dar lugar a que toda una zona de la red se vuelve pasiva.

Para el proceso de aprendizaje, las redes neuronales usan un proceso de *feedback* llamado *backpropagation*. Este proceso consiste en comparar la salida que produce una red con la salida que se supone que debería producir, de esta manera usa el error para ajustar los pesos entre conexiones, desde las capas de salida hasta las capas de entrada. Con el tiempo, *backpropagation* hace a la red aprender, reduciendo el error producido por sus conexiones y haciendo las cosas como se supone que debería.

9.1.2 Descripcion Deep Q-network (DQN)

Deep Q-network [10] fue introducido por el paper *Playing Atari with Deep Reinforcement Learning* publicado por Deepmind en 2013. Normalmente las técnicas de aprendizaje por refuerzo antes mencionadas representan el conocimiento mediante tablas. Esto tiene unas limitaciones computacionales muy grandes. Deep Q-network substituye dicha tabla por una red neuronal que aproxima el conocimiento.

DQN consta de cuatro técnicas con las que consigue superar el aprendizaje inestable:

- **Experience replay**: Esta técnica almacena experiencia incluyendo transiciones, recompensas y acciones que son información necesaria para hacer

funcionar Q-learning y genera *mini-batches* para actualizar la red neuronal. Esta técnica produce los siguiente beneficios:

  - reduce correlación entre experiencia y actualizar DNN (*deep neural network*)
  - incrementa la velocidad de aprendizaje con los *mini-batches*
  - reutiliza transiciones antiguas para evitar que *olvide* información relevante

  En este proyecto se usa *experience replay* para el entrenamiento del modelo de DQN y DDQN. Se entrena en *mini-batches* cada vez que se reinicia un episodio (muerte del jugador o el enemigo). Se hacen 16 *mini-batches* y se entrena usando la potencia de la tarjeta gráfica mediante CUDA.

- **Target network**: Cuando se calcula el error de TD, la función objetivo es cambiada frecuentemente cuando se usan *deep neural networks*. Cuando estas funciones objetivo son inestables, el entrenamiento se vuelve difícil. *Target network* arregla los parámetros de la función objetivo y los remplaza por la última red cada X número de pasos.

- **Capas convolucionales**: Al igual que Deepmind, en este proyecto se trabaja con los píxeles de la pantalla en los métodos DQN y DDQN, por lo que al igual que ellos se trabaja con capas convolucionales en nuestra red neuronal. Las redes convolucionales se diferencian principalmente porque asumen que la entrada son imagenes, lo que nos permite codificar algunas propiedades en la arquitectura. Esto hace que la *forward function* sea más eficiente y reduce de manera considerable la cantidad de parámetros en la red.

- **Skipping frames**: Normalmente los juegos son renderizados a 60 o 30 frames por segundo, pero realmente un jugador no realiza una acción en un juego por cada frame. El agente no necesita calcular los *Q-values* por cada frame, por lo que el saltarse frames es algo natural y al igual que en Deepmind se calcula el Q-value cada 4 frames usando los 4 frames anteriores como entrada. Esto reduce el coste computacional y recolecta más experiencias para el agente. Además se utilizan frames de 100x100 en escala de grises debido a que la mayoría de los píxeles de la pantalla no aportan información y de esta manera nos quedamos solo con la información relevante y reducimos aun más el coste computacional.

9.2 Representación de estados y recompensas

Como se ha comentado anteriormente, para los métodos de DQN y DDQN se ha usado el método de Deepmind que usan los píxeles de la pantalla para aprender. A diferencia de Q-learning y SARSA aqui no se usa una representación tabular del conocimiento y en vez de eso se usa una DNN (*deep neural network*) que recibe

como entrada imagenes que son los píxeles de la pantalla y que representan el estado del juego. Cada 4 frames se procesa uno. Además como se ha explicado anteriormente se reduce la imagen del frame de juego a 100x100 en escala de grises para aumentar la eficiencia y reducir el ruido sobre la información. Se usa un *frame buffer* para almacenar los frames con los que se va jugando al juego y esos mismos se usan para entrenar. Como se ha explicado anteriormente cada vez que termina un episodio se entrena en 16 *mini batches* y se actualiza la red neuronal.

En cuanto a la función de recompensa, se han usado dos distintas. Una de ellas usa distancia y la otra no. Son muy similares a las presentadas para Q-learning y SARSA pero con algunas pequeñas diferencias.

Result: Función recompensa con distancia para DQN y DDQN
reward = 0 diff = previousHP - currentHP **if** *diff mayor 0* **then**
  reward += (diffencia/5)*-0.1;
else
  reward += 0.05;
end
bossdiff = previousHPBoss - currentBossHP **if** *bossdiff mayor 0* **then**
  reward += (diffenciaBoss/5)*0.1;
else
  reward += -0.05;
end
if *distance menor 5* **then**
  reward += 0;
else
  if *distance menor previousDistance* **then**
    reward += 0.25
  else
    reward += -1
  end
  previousDistance = distance
end
return reward;

Algorithm 2: Pseudocódigo función recompensa con distancia para DQN y DDQN

En esta función recompensa con distancia se mantiene la parte de Q-learning respecto a la vida rebajando la cantidad de recompensa ganada y perdida multiplicandolo por un factor de 0.1 para balancearlo con la recompensa por distancia que se calcula a continuación. Se escoge un threshold de distancia donde los agentes estan lo suficientemente cerca para atacar al jefe final a la vez o esquivar sus ataques. Este threshold es 6 unidades de distancia. Si la acción de los agentes hace que esten dentro del threshold no ocurre nada, ahora bien si estan fuera de alcance y la acción los aleja aun más se quita un punto de recompensa y si los acerca se gana 0.25 puntos de recompensa.

9.3 Entrenamiento y resultados

A diferencia de los dos primeros algoritmos, para DQN y DDQN se han probado solamente 2 modelos, uno usando distancia en la función recompensa y otro sin usarla.

9.3.1 Modelo sin distancia

Para el entrenamiento de este modelo se dejó entrenando *9 horas, 43 minutos y 4 segundos*. En este tiempo se produjeron *947* episodios con un total de *15296* pasos. Se derrotó al jefe final 252 veces por lo que en el entrenamiento se obtuvo un 26,6 % de victorias. Además, el episodio con victoria más rápida fue de *22* segundos.

Como podemos comprobar hay un incremento considerable de los resultados al cambiar la representación tabular por una red neuronal. Aun así el agente solo gana la mitad de las veces que juega aproximadamente. El daño promedio por otro lado si es muy positivo y indica que en todos los episodios el enemigo o muere o se queda a muy pocos golpes de morir. No es un comportamiento perfecto pero se podría definir como un comportamiento notable.

Puntuación de muertes: 48

Puntuación de vida: 80

9.3.2 Modelo con distancia

Para el entrenamiento de este modelo se dejó entrenando *20 horas, 39 minutos y 29 segundos*. En ese tiempo se produjeron *463* episodios con un total de *26292* pasos. En ese tiempo se derroto al jefe enemigo 125 veces. Por lo que el ratio de victorias en el entrenamiento ha sido del 26,9 %. Además, el episodio con victoria más rápida fue de *17* segundos.

Este modelo aun siendo peor que el anterior, es mejor que los vistos en SARSA y Q-learning. No tiene un gran comportamiento pero se puede definir como un comportamiento aceptable.

Puntuación de muertes: 44

Puntuación de vida: 65

9.3.3 Explicación resultados

Como se ha dicho anteriormente, los resultados de DQN mejoran claramente SARSA y Q-learning. Si bien es cierto que no podemos catalogar estos resultados como comportamiento experto, empiezan a tener una idea clara de que hacer en cada momento. Se definen algunas estrategias básicas como el esquivar y golpear (hit and run) o el golpear de manera simultanea para máximizar el daño. Posiblemente con más tiempo para entrenar y un hardware más potente y especializado se podrían haber obtenido resultados mejores.

Para el entrenamiento de los modelos se usaron los siguientes párametros: una ϵ variable, empezando en 0.999 y acabando en 0.01.

Capítulo 10

Double Deep Q-Network

10.1 Descripción

Double Deep Q-network (DQN) [18] es una mejora del algoritmo anteriormente explicado Deep Q-network (DQN). La inspiración principal detras de double DQN es que el DQN regular acostumbraba a sobrestimar los *Q-values* de las acciones potenciales dadas un estado dado. Esto podría ser aceptable si todas las acciones fueran sobrestimadas por igual, pero no es el caso. Si el *Q-value* de ciertas acciones subóptimas es más elevado que el de las acciones óptimas al agente le será muy difícil aprender la política ideal. Para corregir este hecho los autores de DDQN proponen el siguiente truco: en vez de coger el máximo de los *Q-values* cuando se esta calculando el valor objetivo de Q (Q-target) en un paso de entrenamiento, usamos nuestra red principal para escoger una acción y nuestra red objetivo para generar el valor objetivo de Q para esa acción. Separando la elección de la acción de la generación del valor objetivo de Q, somos capaces de reducir de manera substancial la sobrestimación, entrenar más rápido y con más seguridad de obtener buenos resultados.

En Q-learning usamos la siguiente fórmula para calcular el valor objetivo de Q (target value for Q):

$$Q - Target = R_{t+1} + Q(S_{t+1}, argmaxQ(S_{t+1}, a; \theta t); \theta t). \tag{10.1}$$

Como hemos explicado anteriormente esta equación sobrestima, por lo que DDQN la cambia a la siguiente equación:

$$Q - Target = R_{t+1} + Q(S_{t+1}, argmaxQ(S_{t+1}, a; \theta t); \theta' t). \tag{10.2}$$

Cambia el origen de donde se escoge la acción que viene de θ ' en vez de θ. En definitiva DDQN es una mejora de DQN donde se usan dos redes neuronales (de ahi el termino double), usamos la *red principal* para escoger la acción a realizar y la segunda red, que llamamos *red objetivo*, se utiliza para calcular el valor del *Q-target*.

10.2 Representación de estados y recompensas

Ya que DDQN es una mejora de DQN, se han utilizado los mismos métodos para la representación de estados usando los píxeles de la pantalla y se han probado las mismas dos funciones recompensa que se han explicado en el capítulo anterior.

10.3 Entrenamiento y resultados

Se han entrenado dos modelos para DDQN, uno que incluye distancia en su función recompensa y otro que solo trabaja con la vida actual de los entes en el juego.

10.3.1 Modelo sin distancia

Para el entrenamiento de este primer modelo se ha dejado entrenando *20 horas, 39 minutos y 29 segundos*. Se han realizado un total de *463* episodios cuya suma total de pasos ha sido de *26292*. En este tiempo el jefe final fue derrotado *125*. Además, se hizo un récord venciendo al jefe final en *17 segundos*. Durante el entrenamiento el agente tiene un ratio de victoria del *26 %*.

Al igual que pasaba en DQN, en DDQN los resultados son mucho mejores que en SARSA y Q-learning. En este modelo de consigue el pico de porcentaje de victorias superando la mitad de victorias por episodio. Prácticamente duplica y triplica los resultados de los algoritmos anteriormente probados (excepto DQN) en cuanto a puntuación de muerte se refiere. En cuanto el daño promedio sigue siendo alto al igual que en DQN.

Puntuación de muertes: 60

Puntuación de vida: 70

10.3.2 Modelo con distancia

Para el entrenamiento de este segundo modelo se ha dejado entrenando *1 día, 9 horas y 9 segundos*. En este tiempo se han realizado *704* episodios con un total de *28030* pasos sumados. En este tiempo se ha vencido al boss *108* veces. Por lo que durante el entrenamiento el agente tiene un ratio de victoria del *15.34 %*.

Si el modelo anterior de DDQN conseguía el pico en porcentaje de victorias este modelo con distancia consigue el pico en daño promedio al jefe enemigo. En este

modelo el enemigo final se queda a un par de golpes de ser eliminado en promedio de todos los episodios jugados.

Puntuación de muertes: 50

Puntuación de vida: 85

10.3.3 Explicación de resultados

Como era de esperar, los resultados de DDQN son los mejores de todos los algoritmos probados. Al igual que con DQN el cambiar la representación tabular a una red neuronal y trabajar desde los píxeles hace que el aprendizaje sea mucho más efectivo. Además, como DDQN es una mejora de DQN era de esperar que diese resultados muy similares o mejores. Al usar dos redes neuronales: una para escoger la acción y otra para calcular la función objetivo, se consigue una mejor eficiencia en el entrenamiento. Aun siendo los mejores resultados, si es cierto que no podemos considerarlos un comportamiento excelente. Aunque los agentes utilizan algunas estrategias de colaboración y esquiva y ataque, no siempre las utilizan y aun les faltaría más tiempo de aprendizaje para llegar a un comportamiento excelente.

Para el entrenamiento de los modelos se usaron los siguientes parámetros: una ϵ variable, empezando en 0.999 y acabando en 0.01.

Capítulo 11

Comparación de resultados

Como bien se ha ido explicando para cada algoritmo, se han probado cuatro algoritmos distintos: Q-learning, SARSA, DQN y DDQN. Y de estos algoritmos se han hecho tres modelos para Q-learning y SARSA y dos modelos para DQN y DDQN, un total de diez modelos. La diferencia entre modelos es clara, el modelo 1 usa como representación de estados y/o función de recompensa la diferencia de vida, el modelo 2 (solo usado en Q-learning y SARSA) utiliza la vida del agente como representación de estados. Por último, el modelo 3 utiliza distancia entre los dos personajes tanto en la representación de estados (Q-learning, SARSA) como en la función de recompensa (DQN, DDQN), este modelo es el único que utiliza información que no es extraída directamente de los píxeles del juego sino de el código interno del juego.

A continuación se mostrarán dos tablas con los resultados de todos los modelos. Tanto de la puntuación de muerte (porcentaje de victorias sobre 100) como la puntuación de vida (daño promedio al jefe enemigo también sobre 100).

	Modelo 1	**Modelo 2**	**Modelo 3**
Q-learning	11	8	7
SARSA	26	20	23
DQN	48	X	44
DDQN	60	X	50

Cuadro 11.1: Tabla de puntuación de muertes

Viendo los resultados vamos a intentar compararlos tanto por modelos, puntuación y algoritmos utilizados.

	Modelo 1	Modelo 2	Modelo 3
Q-learning	57	41	32
SARSA	55	51	54
DQN	80	X	65
DDQN	70	X	85

Cuadro 11.2: Tabla de puntuación de daño promedio

11.1 Daño promedio

Comencemos comparando los resultados por su puntuación de vida. Es decir su promedio de daño al jefe enemigo en cada episodio. Para diferenciar los modelos entre otros se ha asignado un color a cada modelo: azul al modelo 1, verde al modelo 2 y rojo al modelo 3.

11.1.1 Modelo 1

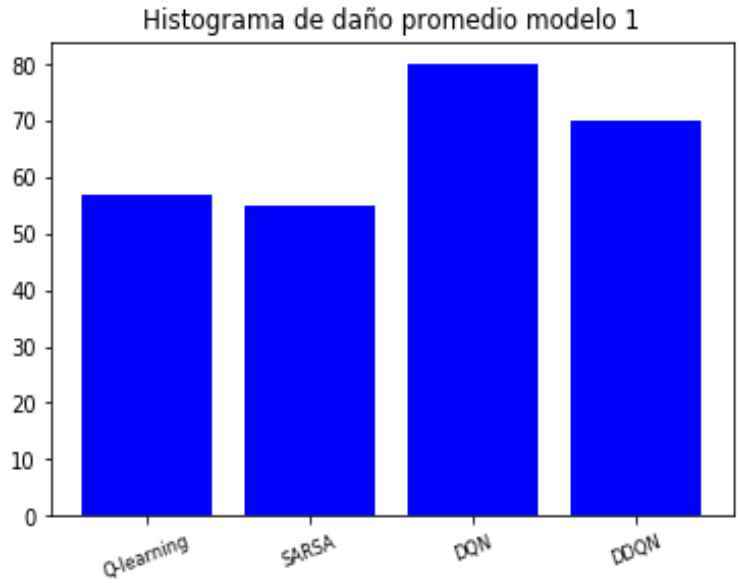


Figura 11.1: Histograma de daño promedio modelo 1

Para el modelo 1 podemos observar que en todos los algoritmos se superan los 50 puntos de daño promedios, por lo que los agentes aprenden a atacar al enemigo. Q-learning y SARSA tienen dos puntuaciones muy similares muy cercanas a 50 puntos. Como era de esperar DQN y DDQN tienen una puntuación más elevada, DQN alcanza una puntuación de 80 y DDQN de 70 ambas muy superiores a las dos primeras. Esto es algo natural debido a la potencia de los algoritmos. Lo

sorprendente es que DQN de mejores resultados que DDQN debido a ser este una mejora del anterior. Esto se puede deber a las circunstancias aleatorias del entrenamiento como los golpes esquivados por el enemigo o el gran problema de el movimiento contra un muro. Otro posible motivo puede ser simplemente la suerte, que en las 100 pruebas que se realizan para puntuar un modelo la suerte haya afectado a DDQN. De todas maneras podemos concluir que los resultados de DQN y DDQN son favorables. El modelo 1 al ser representado por la diferencia de vida entre los agentes y el enemigo, tiene en cuenta tanto el recibir daño como el hacerlo. Por lo que el balance entre el daño que inflinge (puntuación de vida) y el daño que recibe es adecuado.

11.1.2 Modelo 2

Figura 11.2: Histograma de daño promedio modelo 2

El modelo 2 solo existe para Q-learning y SARSA por lo que poca comparación se puede hacer. Los resultados para Q-learning no llega a 50 puntos quedandose en 40, en SARSA la putuación es de 50. Es curioso porque en el modelo 1, Q-learning era mejor que SARSA pero en el modelo 2 pasa lo contrario. Además que los dos resultados bajan respecto el modelo anterior. Se podría decir que esto pasa porque en el modelo 2 se tiene en cuenta solo la vida de los agentes y no la del enemigo. Por lo que se prioriza el no recibir daño al hacer daño. Esto en una batalla contra un jefe final donde solo hay dos posibilidades: morir o matar, es muy malo. Si los agentes no priorizan el hacer daño al enemigo, este al tener ataques más poderosos los acabará matando la mayoria de veces.

11.1.3 Modelo 3

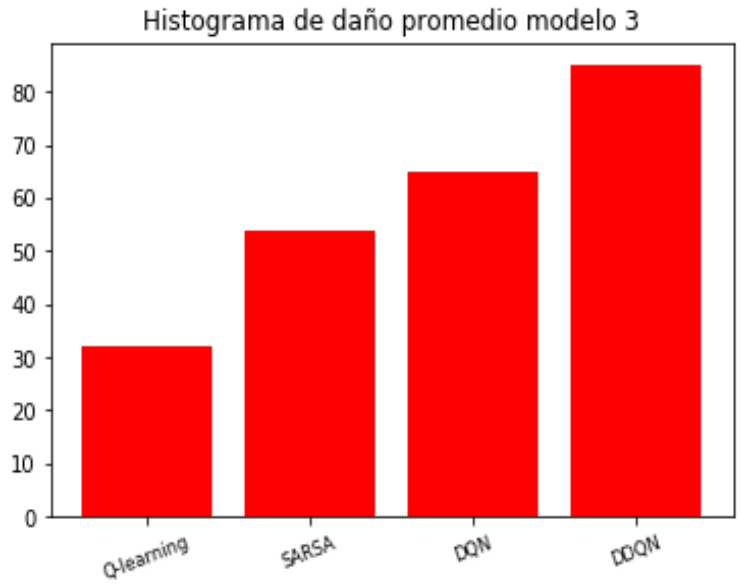


Figura 11.3: Histograma de daño promedio modelo 3

El modelo 3 es una escalera creciente, al igual que en el modelo 1, DQN y DDQN tienen resultados muy superiores a Q-learning y SARSA. En este caso además, Q-learning tiene un resultado muy negativo haciendo una diferencia con SARSA de más de 20 puntos. El de SARSA por otro lado se mantiene rondando los 50 puntos al igual que en el modelo 1 y el modelo 2. Por otro lado DQN y DDQN tienen resultados mucho mejores, sobretodo DDQN que casi triplica el resultado de Q-learning con 85 puntos. En este modelo se alcanza el pico de daño promedio con DDQN, el enemigo se queda a muy pocos golpes de la muerte en todos los episodios que se juegan. DQN por otro lado es ligeramente superior a SARSA y claramente inferior a DDQN. En este modelo es donde más se nota la mejora de DQN respecto a DDQN, aunque como se ha comentado anteriormente también puede haber influido la suerte tanto en el entrenamiento como en las pruebas al puntuar. Este modelo también posee el peor resultado teniendo Q-learning 30 puntos.

En el modelo 3 se utiliza la distancia como representación, y se potencia que los dos agentes estén a menos de distancia 5. Esto provoca que los algoritmos menos potentes tengan resultados más pobres debido a que el enemigo final es capaz de eliminarlos más fácilmente si están juntos y no esquivan a tiempo sus ataques en área. Por otro lado en los algoritmos más potentes que usan redes neuronales, el rendimiento es extraordinario y supera al resto de los modelos. Esto es debido a que se consigue un balance de estar cerca pero a la vez esquivar los ataques del jefe enemigo mientras se ataca. Por este motivo el modelo 3 es el modelo con el pico más alto de daño promedio (DDQN).

11.1.4 Todos los modelos

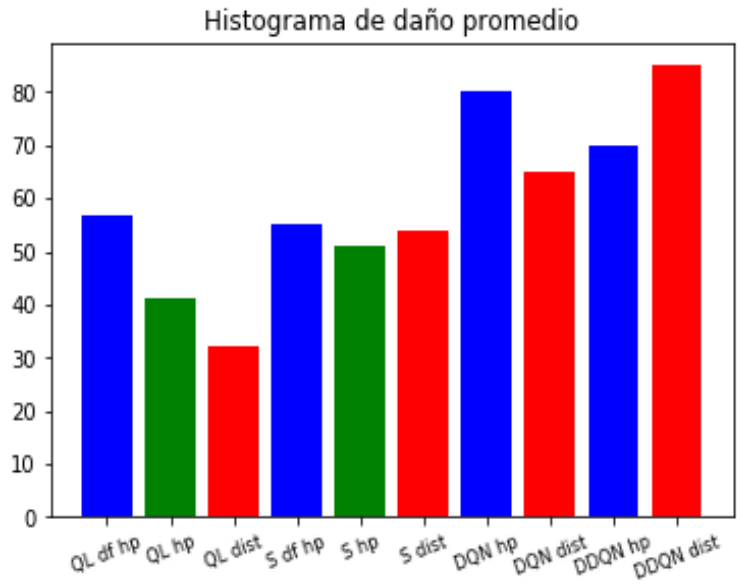


Figura 11.4: Histograma de daño promedio

Como se ha explicado anteriormente, se puede ver claramente los algoritmos DQN y DDQN son para todos los modelos los que poseen mejores resultados tal y como era de esperar. El peor resultado ha sido el modelo 3 de Q-learning con 30 puntos y el mejor ha sido DDQN de también el modelo 3. Aún poseyendo los *outliers*, el modelo 1 es superior al modelo 3 si promediamos todas sus instancias. El modelo 1 tiene un promedio de 65.9 puntos de daño contando todos los algoritmos mientras que el modelo 3 tiene un promedio de 59 puntos de daño. El modelo 2 es claramente inferior al poseer solo los algoritmos menos potentes con un promedio de 46 puntos de daño. Con todos los datos sobre la mesa podemos pues concluir que el mejor modelo teniendo en cuenta el daño inflinjido por los agentes en todos los algoritmos es el primer modelo. Aunque como ya se ha dicho el mejor resultado individual es DDQN en el tercer modelo.

11.2 Porcentaje de victorias

Ahora se procederá a comparar los resultados por su puntuación de muerte. Es decir su porcentaje de victorias sobre el jefe enemigo. Para diferenciar los distintos modelos se han usado los mismos colores que en el apartado anterior.

11.2.1 Modelo 1

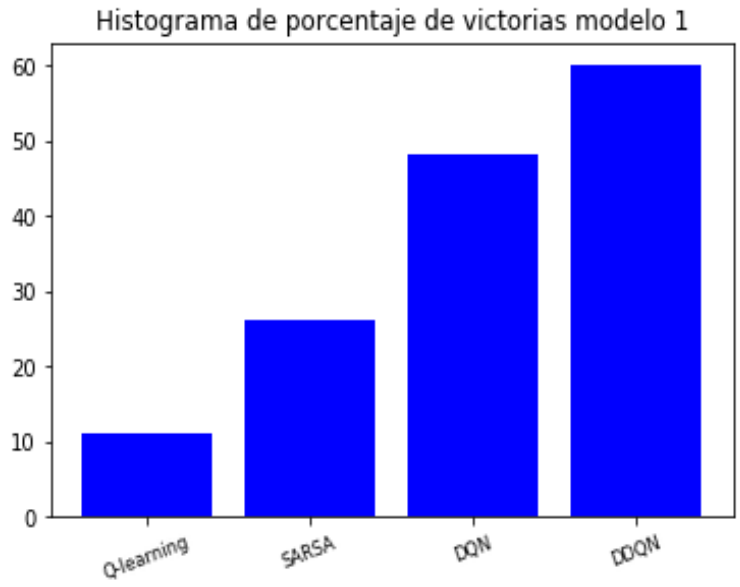


Figura 11.5: Histograma porcentaje de victorias modelo 1

En el modelo 1 empezamos con una escalera, Q-learning tiene unos resultados muy pobres con una puntuación de 11, le sigue SARSA con una puntuación de 26 también pobre. Como pasaba con el daño, aquí los algoritmos menos potentes también poseen resultados mucho más pobres. Cuando llegamos a DQN es cuando nos empezamos a acercar a la mitad de victorias por número de episodios con 48 puntos. Y por último tenemos DDQN con el mejor resultado de todos los modelos, 60 puntos. Los resultados con los dos primeros algoritmos son muy pobres y con los más potentes son decentes. Un 60 % de victorias no es un comportamiento excelente pero si notable.

11.2.2 Modelo 2

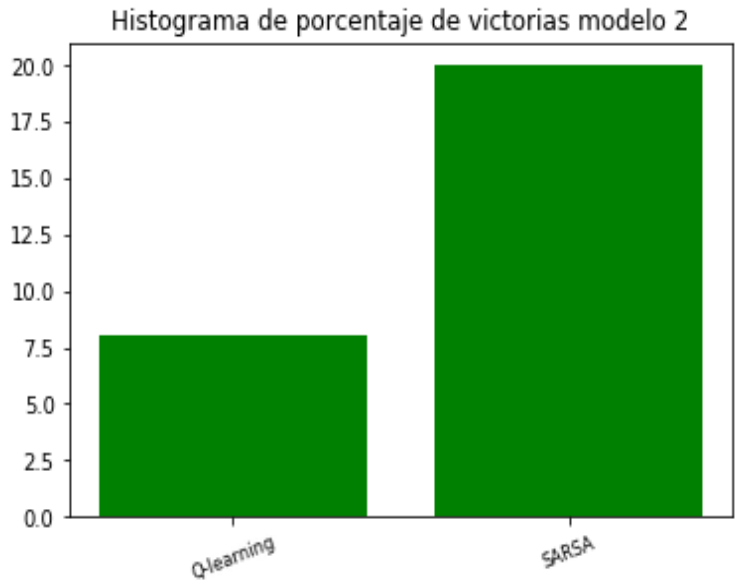


Figura 11.6: Histograma porcentaje de victorias modelo 2

En el modelo 2 solo se usan los algoritmos menos potentes y por ello los resultados de este son muy pobres. Q-learning tiene un resultado pésimo de 8 puntos y SARSA aun siendo malo duplica a Q-learning con 20 puntos. Al igual que con el daño, el porcentaje de victorias es muy malo en este modelo, ambas puntuaciones estan relacionadas ya que si los agentes no inflinjen daño, entonces no podran vencer al jefe enemigo. Los motivos por los que este modelo es tan malo son los mismos que los de la puntuación de daño. El agente solo se preocupa de no recibir daño y por ende no acaba con el enemigo y muere.

11.2.3 Modelo 3

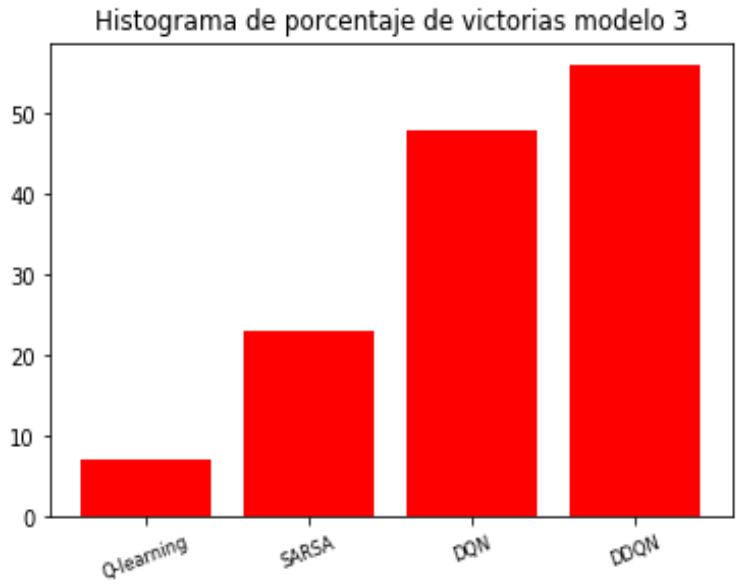


Figura 11.7: Histograma porcentaje de victorias modelo 3

En el modelo 3 volvemos a tener una escalera, el resultado de Q-learning es el peor resultado de todos con 7 puntos, SARSA lo triplica con 23 puntos. Y una vez más los algoritmos más potentes ganan por mucho. DQN se queda cerca de la mitad de victorias por número de episodios con 44 puntos y DDQN alcanza ese objetivo. Aunque no podamos decir que los resultados de los algoritmos potentes sean muy buenos, son aceptables. En DDQN se gana el mismo número de veces que se pierde. Por otro lado, en Q-learning tiene unos resultados pésimos y SARSA se mantiene cerca de los 20 puntos sin pena ni gloria. Este modelo es otra demostración de que algoritmos son mejores y una muestra de que aun inflingiendo mucho daño, la victoria no siempre se consigue.

Con el uso de distancia entre personajes, los agentes incrementan el daño que se hace al enemigo con algunos algoritmos, pero también se incrementa el riesgo a recibir un impacto doble con un ataque en área del enemigo que ponga la partida en jaque. Aun así un balance entre distancia y diferencia de vida es siempre la mejor opción.

11.2.4 Todos los modelos

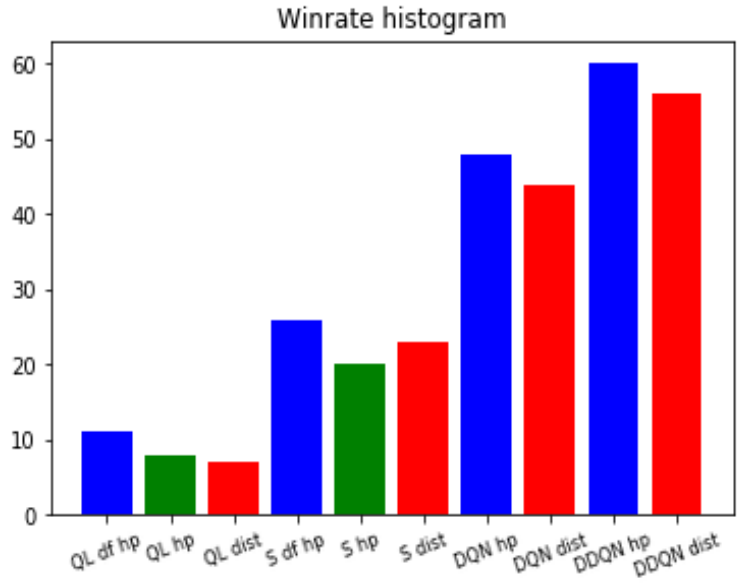


Figura 11.8: Histograma puntuación porcentaje de victorias.

Recapitulando, lo que queda claro es que el peor algoritmo con diferencia en esta puntuación es Q-learning. Tiene puntuaciones pésimas en todos los modelos. SARSA en cambio aun siendo un algoritmo poco potente y teniendo malos resultados gana el 25 % de las veces aproximadamente. DQN se queda rondando los 50 puntos y DDQN los sobrepasa. Tras comparar todos los resultados con esta puntuación podemos concluir que el mejor modelo es el primero con un promedio de 36.25 puntos contando todos los algoritmos. El segundo mejor modelo es el tercero con 31 puntos de promedio. Por último, una vez más, el segundo modelo que solo dispone de los peores algoritmos es el que tiene peores resultados con un promedio de 14 puntos. El mejor resultado individual es DDQN con el primer modelo.

Siendo el primer modelo el mejor en ambas puntuaciones podemos concluir que este es la mejor representación para nuestro agente.

Capítulo 12

Problemas y estrategias

12.1 Mayores problemas

A lo largo de todo el entrenamiento de los agentes, se han producido algunos problemas que han entorpecido de manera severa el aprendizaje del mismo. A continuación se explicarán con detalle.

12.1.1 Movimiento contra muro

Hay acciones que para el ser humano parecen lógicas por instinto, pero a veces no nos damos cuenta que algo lógico para nosotros, es así gracias a nuestra gran capacidad de computo natural (cerebro) y que para otras entidades pensantes, como por ejemplo la inteligencia artificial que se ha desarrollado en este proyecto, es difícil y se estanca en algo que para nosotros es trivial. Este es el caso de el mayor problema que se ha tenido en este proyecto y que ha dado pie a muchos dolores de cabeza y estrategias.

Como se ha explicado en la sección 4.1.2, el juego funciona con micro turnos de manera que el jefe enemigo realiza una acción inmediatamente después de que el jugador principal la haga (no el compañero). El problema viene cuando se realiza una acción de movimiento hacia una dirección en la que no se puede avanzar porque hay un muro. Cuando esto pasa, el juego interpreta que esta acción es inválida y la ignora por lo que el jugador no se mueve del sitio y en consecuencia el sistema de micro turnos se estanca y no le toca al enemigo. El problema es que para el algoritmo, la acción si cuenta como realizada y si en el entrenamiento por ejemplo, el algoritmo decide que el mejor movimiento es un movimiento contra un muro, este lo realizará constantemente hasta que se de el caso donde el agente realiza la acción aleatoria en vez de usar la política. El agente no cambia de estado debido a que no hay acciones reales en el juego ni por su parte ni del enemigo y por ese

motivo realiza la misma acción en bucle. Cuando esto pasa el compañero es el único que puede reaccionar, pues aunque el protagonista esta estancado, si la acción del compañero es válida se realizará. Esto ha generado retrasos muy elevados en el entrenamiento debido a que el agente se pasaba horas realizando la misma acción hasta salir de la situación contra el muro. Este problema ha afectado mucho más a los algoritmos menos potentes, como Q-learning o SARSA, que a los algoritmos más potentes que usan redes neuronales. Estos últimos como se explicará más adelante en este capítulo idearon una estrategia que se aprovechaba de este problema para explotarlo como un *glitch*.

Para intentar solucionar el problema se decidió crear una variable de *estancamiento* que determina cuantos turnos seguidos pasan realizando la misma acción. Cuando esta variable supera un umbral bastante alto se decide que el agente esta estancado y se realizan acciones contrarias al movimiento que provoco el estancamiento, hasta que se deje de repetir dicho movimiento. Es una solución que choca con el flujo del entrenamiento pero que sirve para que el agente aprenda de manera más rápida y natural.

Es curioso que un problema que para un jugador humano sería trivial para una máquina puede ser un autentico quebradero de cabeza. Aun así, se han sacado cosas positivas de este problema.

12.1.2 Limitación temporal

Como se planteó al inicio de este proyecto, los agentes entrenados por aprendizaje por refuerzo requieren una cantidad de tiempo entrenando muy elevada. Al ser un proyecto de tiempo y hardware limitado los resultados no llegan en ningún momento a comportamiento superhumano. Si se disponiera de máquinas con las que entrenar durante semanas, es muy posible que los algoritmos más potentes como DQN o DDQN hubieran conseguido resultados mejores.

12.1.3 Daño reducido del compañero

Como se explica en la sección 4.1.1, el daño del compañero es inferior al del jugador principal. Esto se debe a que no se aplica el bonus de daño de los objetos equipados por el jugador al compañero, debido a que no los lleva equipados el mismo. Este hecho ha influenciado el entrenamiento del agente priorizando que el que hiciese el daño fuese el personaje principal más que el compañero. Esto podía provocar que en ocasiones el compañero se alejase mucho de la acción y que el compañero principal se tuviera que encargar solo del enemigo.

Este problema a despistado ligeramente el proceso de aprendizaje mientras el agente estaba entrenando, pero al ser algo intrínseco del juego se decidió dejar

como estaba. De esta manera además se consiguen estrategias distintas a si los dos personajes tuvieran el mismo daño.

12.2 Estrategias

Al haber probado todos los modelos y ver los resultados de los mismos luchando contra el jefe enemigo, los modelos han adquirido estrategias a seguir para intentar vencer. De todas las estrategias que han seguido, se han destacado tres que utilizan la cooperación entre ambos personajes de alguna forma y se explicarán a continuación.

12.2.1 Hit and run

Golpear y correr (hit and run) es una estrategia muy típica en los juegos, también conocida como *kitear* consiste en golpear al enemigo y inmediatamente ponerse a correr en dirección opuesta a el para esquivar sus ataques. En *Pixel Dungeon* los ataques que poseen nuestros personajes son todos *melee* o de corta distancia por lo que se han de acercar para golpear. La estrategia es muy simple: cuando el enemigo carga el ataque, el agente se acerca le da un golpe y se aleja antes de que el ataque sea realizado. La manera más eficiente para esto es realizar movimientos en diagonal continuos acercándose, golpeando y alejándose.

Esta estrategia la han adquirido los agentes con todos los algoritmos, incluso los algoritmos menos potentes la utilizan en alguna ocasión puntual.

12.2.2 Ataque sincronizado

Una estrategia sencilla para un jugador humano pero no tanto para una máquina, si los dos personajes atacan a la vez al jefe enemigo, este recibirá más daño y solo perseguirá a uno de ellos excepto si realiza un ataque en área.

Esta estrategia también ha sido adquirida por todos los algoritmos aun siendo en algunas ocasiones.

12.2.3 Glitch exploit

A lo largo de la vida de un videojuego, llega un punto donde los jugadores se conocen todos los detalles de un juego y en ocasiones idean estrategias a su favor utilizando un error del videojuego. Este tipo de errores se llaman *glitches* y en

muchas ocasiones aprovecharse de ellos otorga unos beneficios que costaría mucho más conseguir de manera legal en el juego.

Explotar este tipo de *glitches* puede ser considerado una estrategia un poco sucia, pero si esta en el juego es válida. Lo curioso es que el agente a raíz del problema del movimiento contra el muro 12.1.1, ha conseguido idear una manera de beneficiarse de ese problema y explotarlo a su favor.

La estrategia consiste en que el jugador principal realiza constantemente un movimiento contra un muro de manera que el turno del enemigo no llegue nunca y mientras mantiene al enemigo quieto sin moverse debido a este *glitch* el compañero le viene por la espalda y le empieza a atacar sin descanso. Esta estrategia cuando se utiliza suele ser efectiva más del 80 % de las veces.

Esta estrategia al ser bastante compleja solo ha sido adquirida por los algoritmos más potentes (DQN y DDQN) y se realiza en situaciones muy especificas.

Capítulo 13

Conclusión

Tras todo el trabajo realizado, podemos concluir que los resultados de los agentes han tenido un éxito aceptable. Los algoritmos más potentes consiguieron resultados aceptables para una batalla contra un jefe. Tras todas las pruebas y modelos probados se ha demostrado, que efectivamente como era de esperar los algoritmos que utilizan una representación tabular son muy inferiores a los que utilizan redes neuronales. Con este trabajo también se ha demostrado la gran fuerza y utilidad del *deep learning* trabajando directamente con imágenes, píxeles de la pantalla en este caso.

Las distintas representaciones utilizadas en cada uno de los algoritmos han servido para comprobar la importancia de una buena representación de la información en este tipo de proyectos y lo mucho que influyen aún utilizando los mismos algoritmos. En el caso particular de este proyecto la mejor representación de estados ha sido usando los frames del juego pasados a 100x100 en escalas de grises y usar como función recompensa la diferencia de vida del jugador respecto al enemigo.

En cuanto a los algoritmos las principales conclusiones son las siguientes:

- Los algoritmos que usan redes neuronales no han conseguido un comportamiento superhumano pero con más tiempo y hardware no se descarta que se consiguieran resultados cercanos al comportamiento superhumano
- Los algoritmos como SARSA y Q-learning no son los suficientemente potentes para llegar a alcanzar

Tras realizar este proyecto donde se agregaba un compañero y se hacía cooperar con el jugador principal, se han entendido las grandes dificultades de la cooperación. Se usan multiagentes al entrenar a el compañero y al jugador por separado, pero que tratan de conseguir el mismo objetivo. Para conseguir una cooperación correcta es necesario utilizar los algoritmos más potentes y usar una función recompensa adecuada, que permita descubrir que la mejor manera de vencer sea

cooperando. Se ha conseguido esto último con los mejores modelos donde los agentes utilizan las estrategias de cooperación explicadas anteriormente.

Una importante conclusión que cabe destacar es que escenarios más complejos y con menos acción aun son complicados de abordar con aprendizaje por refuerzo,observandose un funcionamiento mucho mejor cuando se usa en tareas que reciben recompensas tan pronto sea posible. Esto se ha podido comprobar comparando el comportamiento en escenarios de batalla contra un jefe con escenarios con tareas más generales, como jugar de manera habitual al juego. Donde el árbol de expansión de las posibilidades de juego es exponencial, más aun si contamos con dos agentes que nos multiplican las opciones. En mi opinión, queda un largo camino en el aprendizaje por refuerzo para que consiga abordar problemas más generales como Pixel Dungeon en su plenitud junto a un compañero.

13.1 Opinión personal

Trabajar en este proyecto ha sido una gran experiencia personal. A lo largo del grado he ido adquiriendo mucho interés en el campo de la inteligencia artificial y la única rama que no había tocado era el aprendizaje por refuerzo. El poder aprender y profundizar sobre una rama tan interesante y diferente ha sido muy gratificante. Además, por qué obviarlo, poder combinarlo con los videojuegos, que han sido mi pasión desde antes de que tuviera conciencia, ha sido un incentivo más en el desarrollo de este trabajo.

El verme capaz de coger el código fuente de un videojuego y poder amoldarlo a mis intereses del proyecto ha sido muy enriquecedor y me ha permitido pasar por diferentes adaptaciones hasta conseguir el escenario que deseaba. Además, he tenido la oportunidad de obtener amplios conocimientos sobre aprendizaje por refuerzo: desde sus características, hasta cuatro algoritmos con distintas complejidades. He podido implementar dos algoritmos desde cero y otros dos usando librerías de apoyo.

Este trabajo también ha sido un ingente ejercicio de investigación: entender un campo nuevo de la inteligencia artificial, entender el porque los agentes actúan como actúan, inventar representaciones y funciones de recompensa. Muchas de estas decisiones se han tomado por prueba y error con el objetivo de mejorar el comportamiento de los agentes y entender más la dificultad de la cooperación.

En definitiva, a nivel personal, estoy muy satisfecho con los resultados obtenidos, que aún constatando la gran dificultad para que estos algoritmos respondan adecuadamente, los he conseguido implementar de forma satisfactoria para hacer colaborar al compañero con el personaje principal.

Capítulo 14

Trabajo futuro

Como trabajo futuro se podrían realizar las siguientes tareas.

- **Probar otros algoritmos**: Se podrían probar algoritmos ya existentes como: Path Consistency Learning, Dueling DQN o Proximal Policy Aproximation, este último se comenzó a implementar pero al final se decidió optar por los algoritmos escogidos. En mi opinión, es difícil que alguno de estos algoritmos diesen mejores resultados que DDQN pero sería interesante ver como funcionan.
- **Ampliar repertorio de jefes finales**: Se podría probar estas mismas técnicas que se han utilizado para derrotar a Goo para intentar vencer a otros jefes del juego. Viendo como afecta el nivel de dificultad de estos nuevos escenarios sobre las técnicas utilizadas.
- **Aumentar el número de compañeros**: Se podría intentar utilizar los conocimientos de este proyecto para crear a un segundo o tercer compañero y ver que estrategias y posibilidades dan jugando juntos. Seria darle un nivel de dificultad más a la cooperación.

Bibliografía

[1] A Beginner's Guide to Deep Reinforcement Learning, Skymind, A.I Wiki, https://skymind.ai/wiki/deep-reinforcement-learning

[2] Deepmind, https://deepmind.com/

[3] Deep Reinforcement Learning, Deepmind, David Silver, https://deepmind.com/blog/deep-reinforcement-learning/

[4] Artificial Intelligence and Games http://gameaibook.org/

[5] Reinforcement Learning: An Introduction http://incompleteideas.net/book/the-book-2nd.html

[6] Dendi vs. OpenAI at The International 2017, Dota2, https://www.youtube.com/watch?v=wi0opO9jTZw

[7] OpenAI 5v5 vs Dota TOP 0,05 % players — FIRST game vs pro team, https://www.youtube.com/watch?v=lc1fl5bdZdA

[8] The OpenAI Dota 2 bots just defeated a team of former pros, https://www.theverge.com/2018/8/6/17655086/dota2-openai-bots-professional-gaming-ai

[9] Observe and Look Further: Achieving Consistent Performance on Atari, https://arxiv.org/pdf/1805.11593v1.pdf

[10] Playing Atari with Deep Reinforcement Learning, https://arxiv.org/pdf/1312.5602v1.pdf

[11] OpenAI Five, https://www.youtube.com/watch?v=eHipy_j29Xw

[12] RL Course by David Silver, https://www.youtube.com/watch?v=2pWv7GOvuf0&t=1s

[13] Github: awesome-rl https://github.com/aikorea/awesome-rl

[14] Github: DeepLearningFlappyBird
https://github.com/yenchenlin/DeepLearningFlappyBird

[15] Github: Shattered Pixel Dugeon Gtx
https://github.com/00-Evan/shattered-pixel-dungeon-gdx

[16] Github: Coop Shattered Pixel Dungeon gtx
https://github.com/Dartemiss/Coop-Shattered-Pixel-Dungeon-gtx

[17] Neural networks explained
https://www.explainthatstuff.com/introduction-to-neural-networks.html

[18] Deep Reinforcement Learning with Double Q-Learning
http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/download/12389/11847